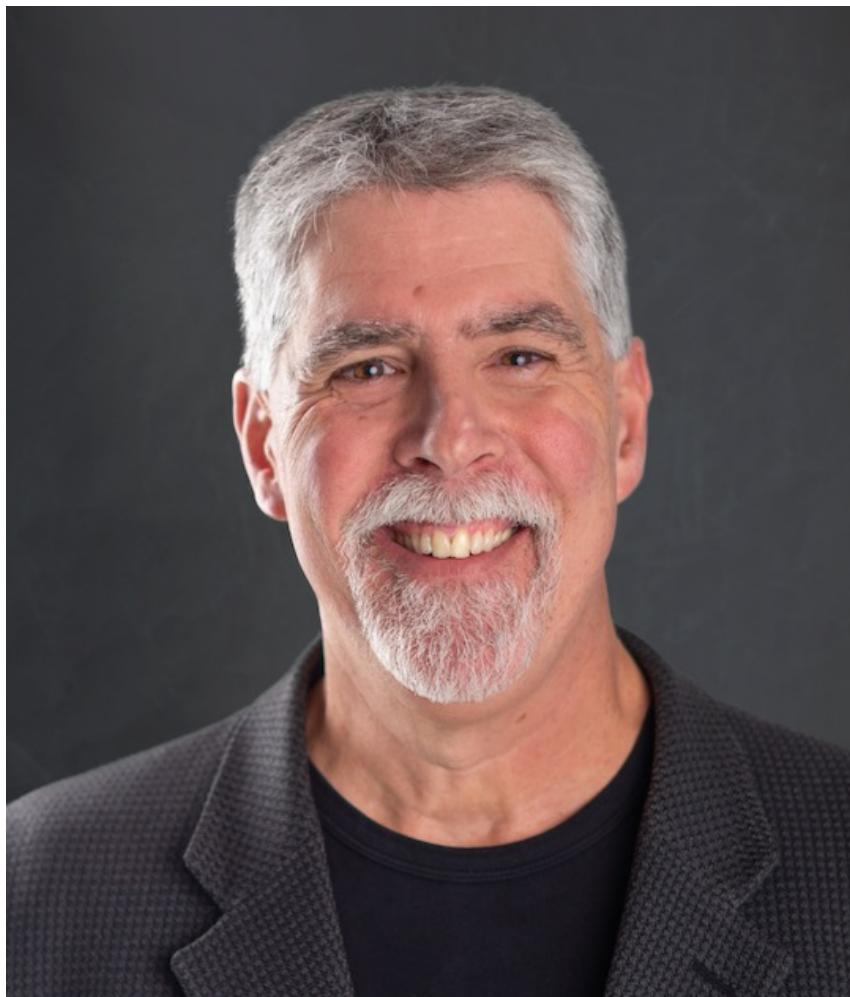




# Mastering Patterns in Event-Driven Architecture



**Mark Richards**

**Independent Consultant**

**Hands-on Software Architect / Published Author**

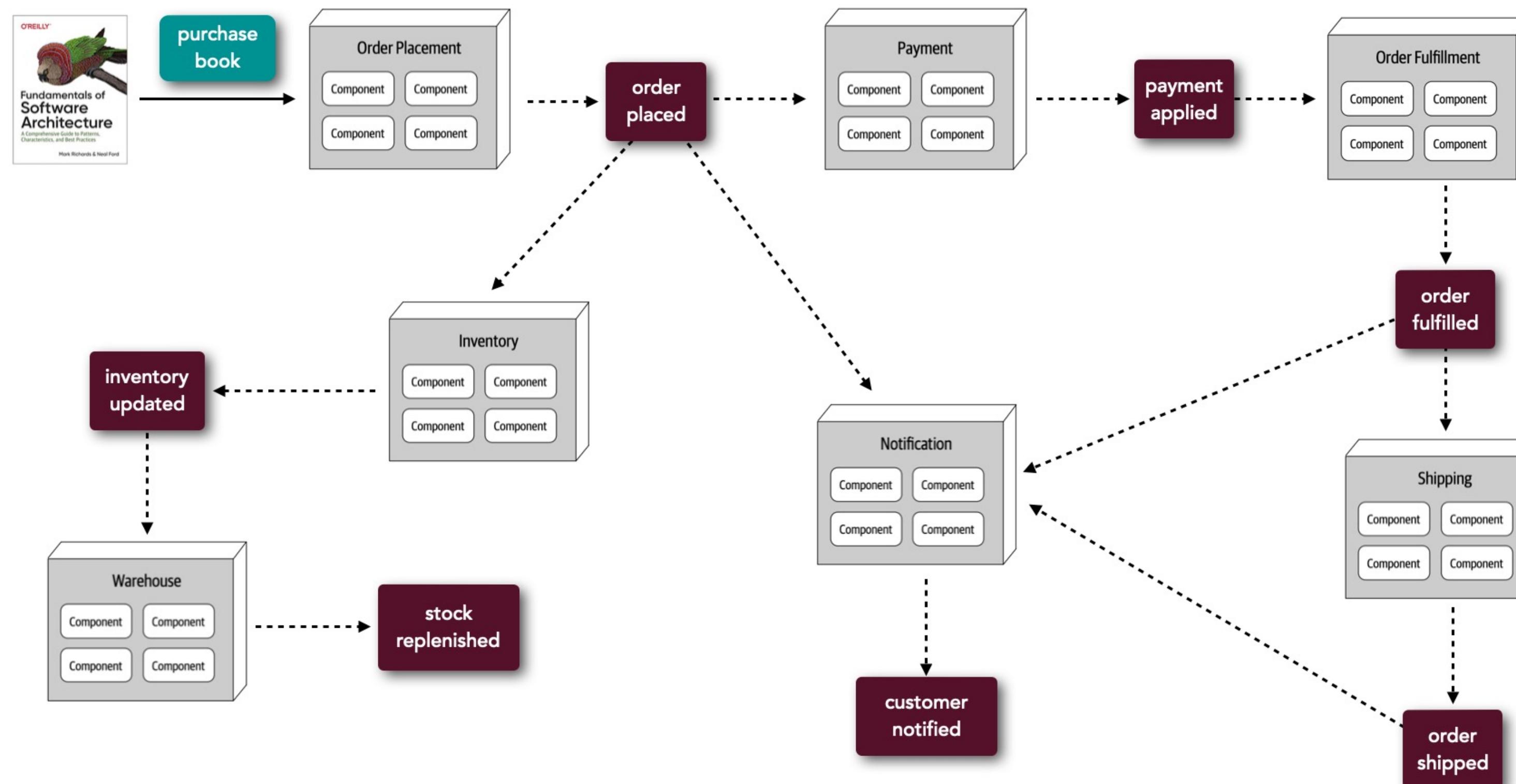
**Founder, DeveloperToArchitect.com**

**<https://www.linkedin.com/in/markrichards3>**

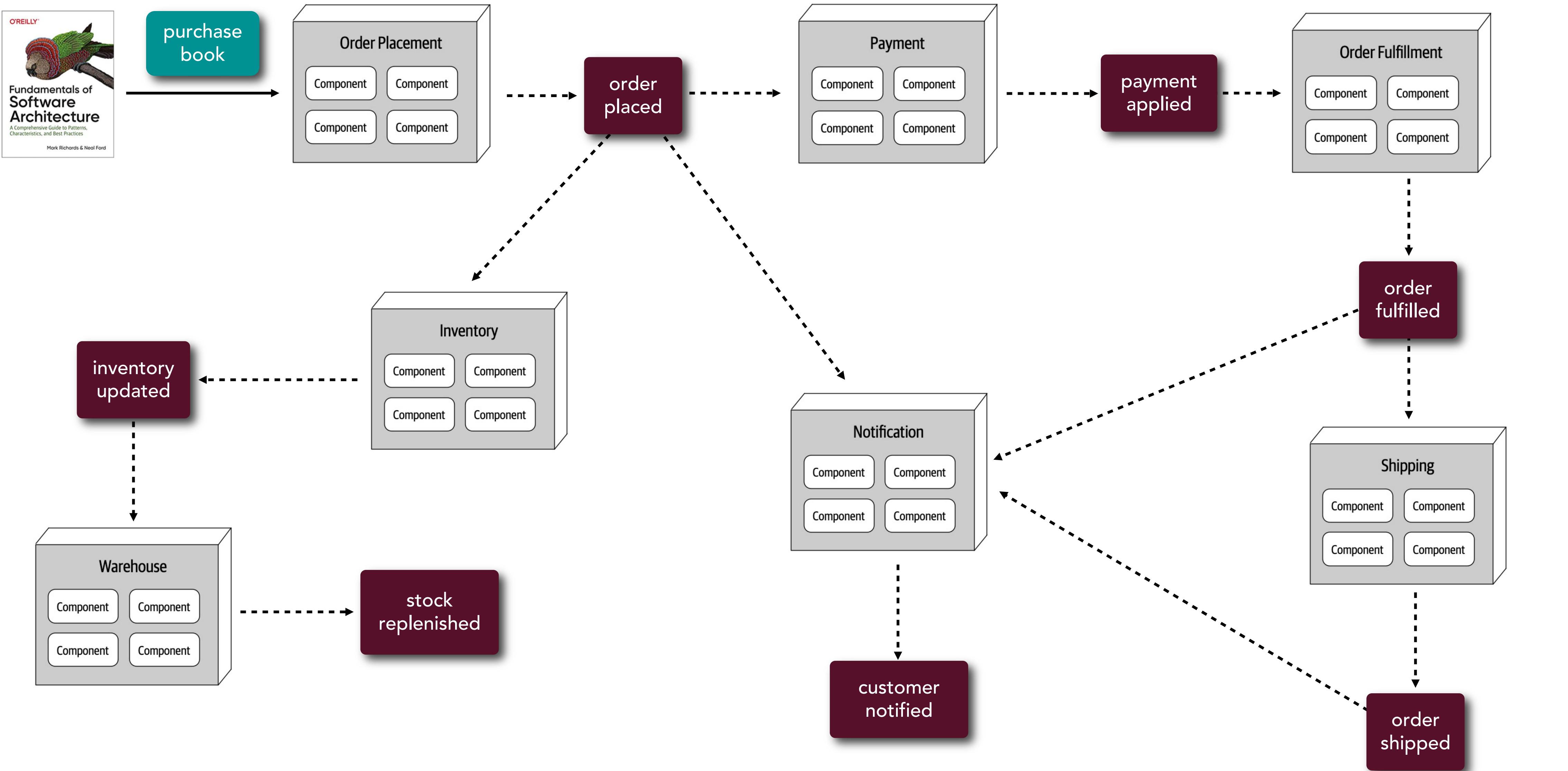
**@markrichardssa**

# event-driven architecture

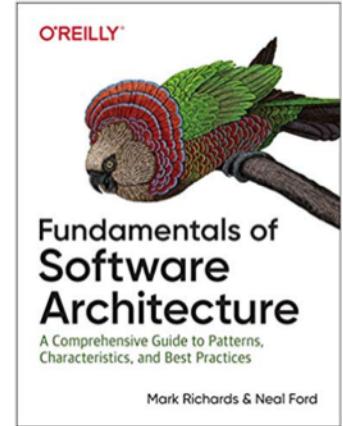
*architecture style that leverages asynchronous processing to trigger and respond to events within the system*



# event-driven architecture



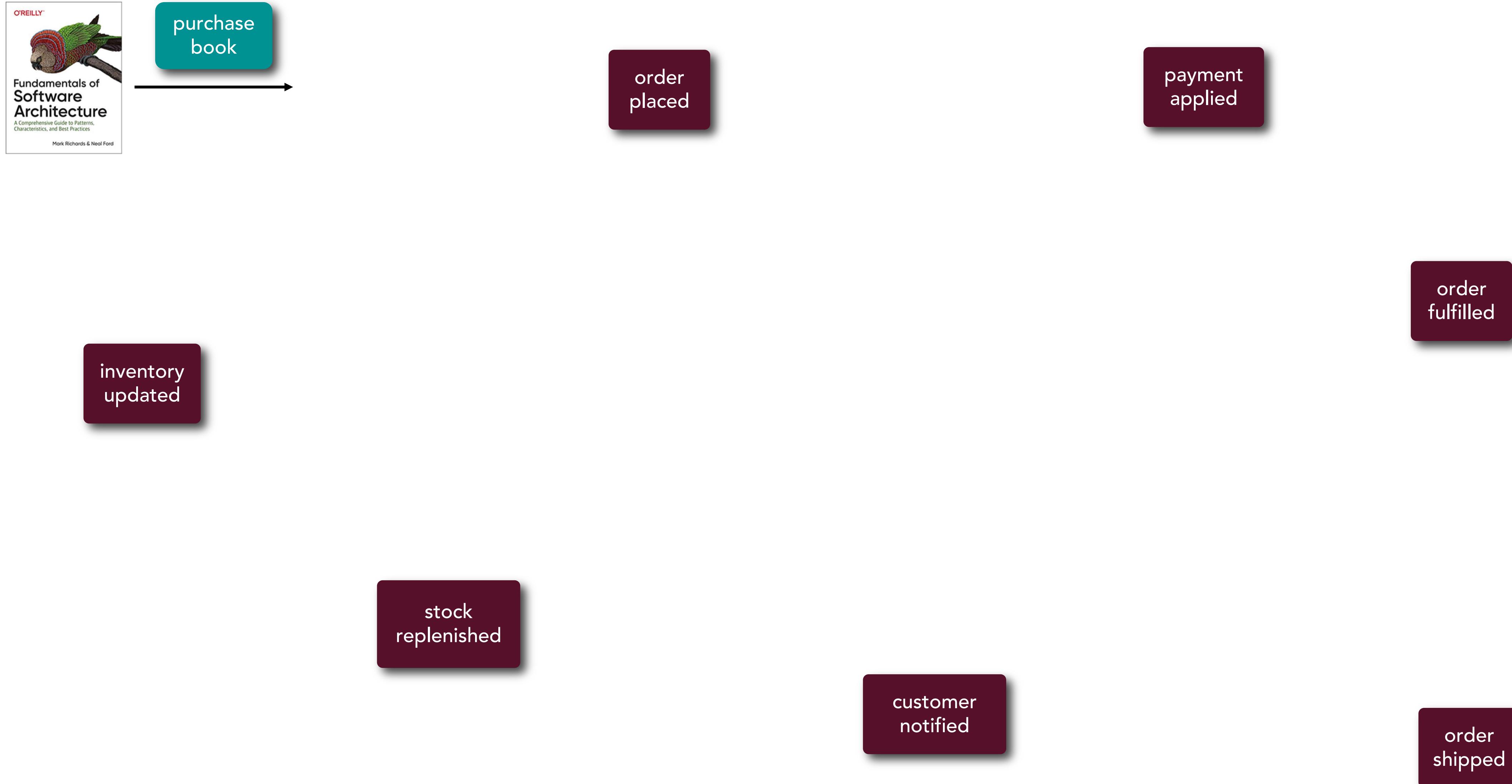
# event-driven architecture



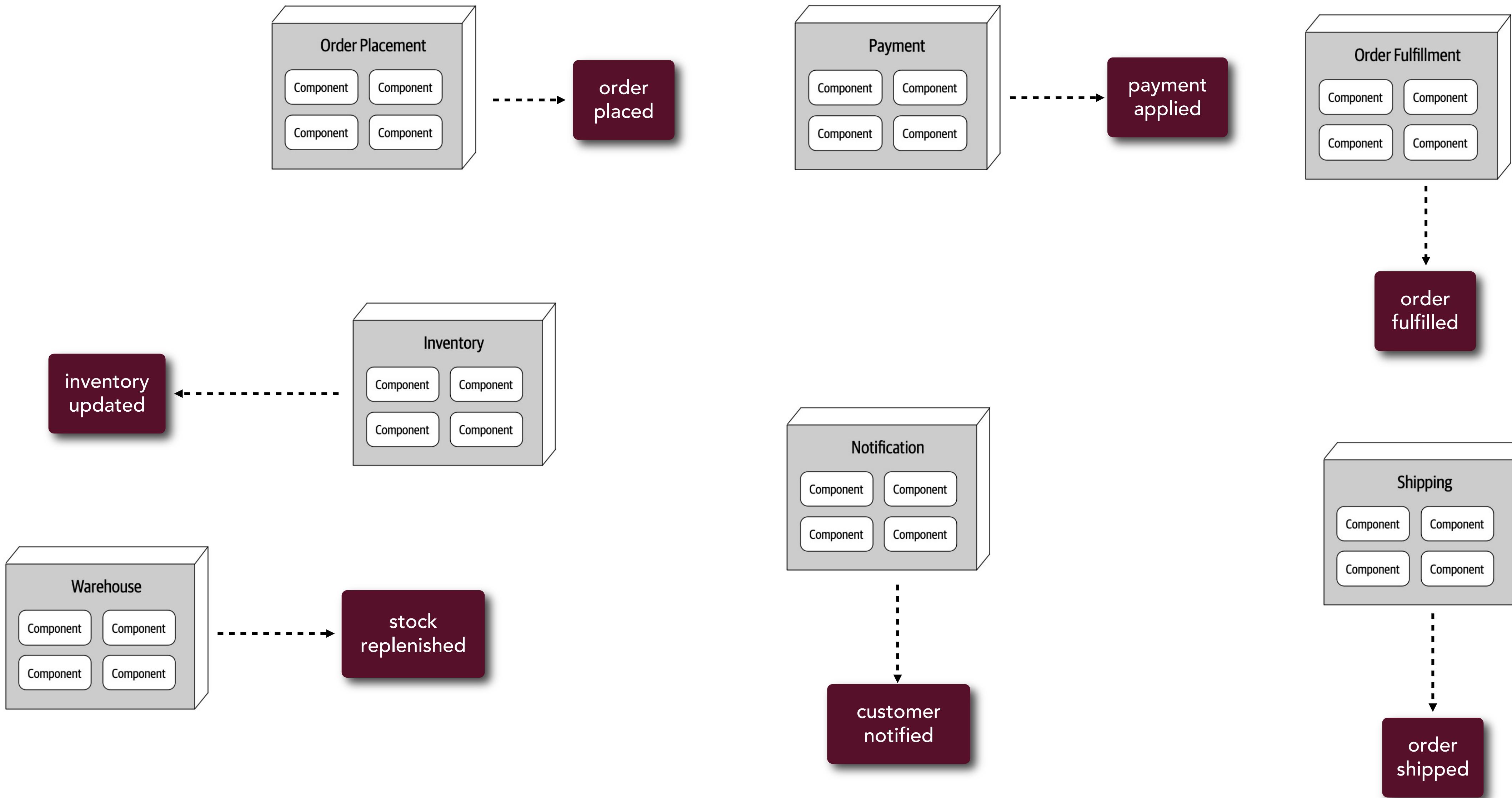
purchase  
book



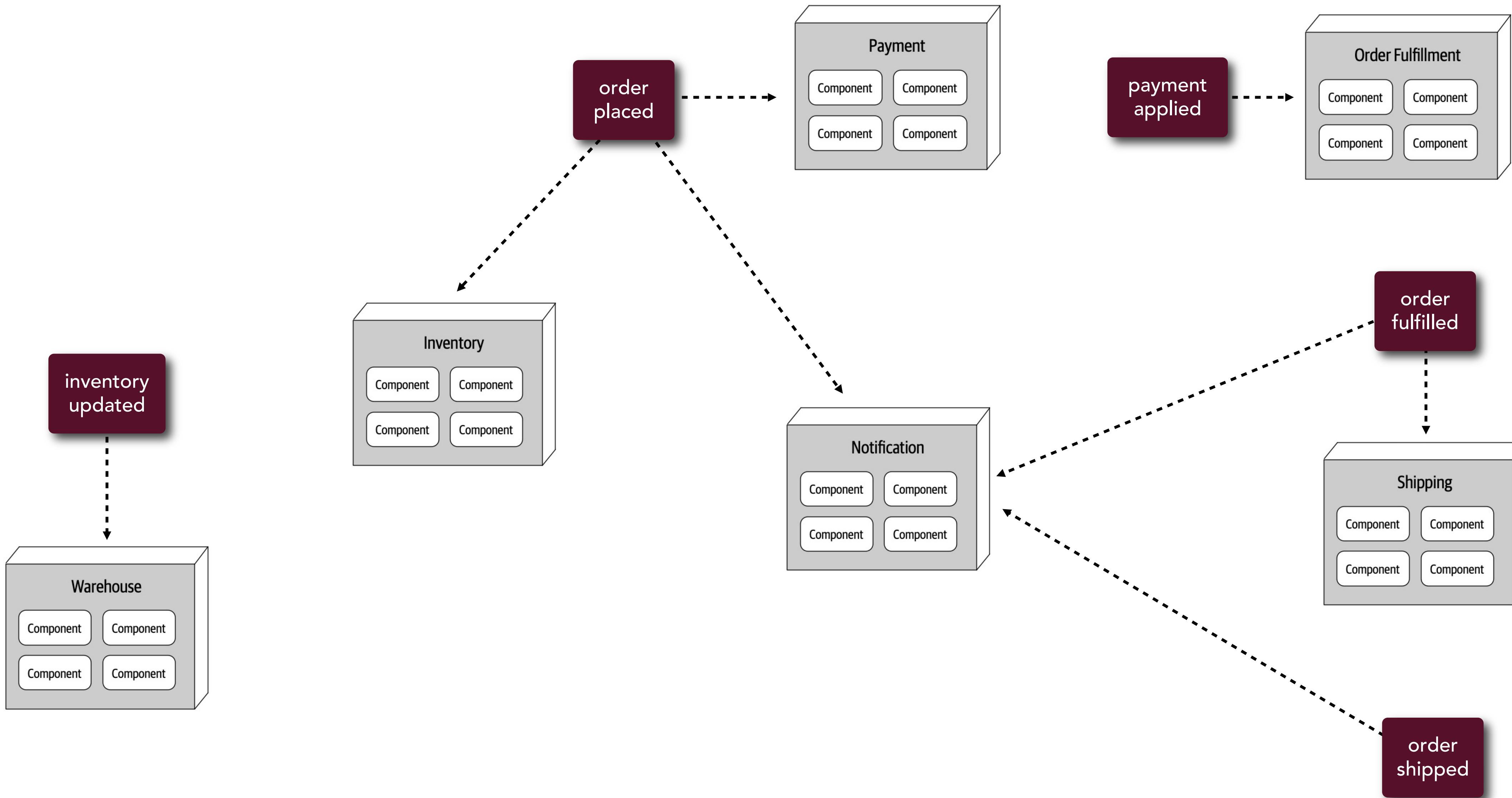
# event-driven architecture



# event-driven architecture

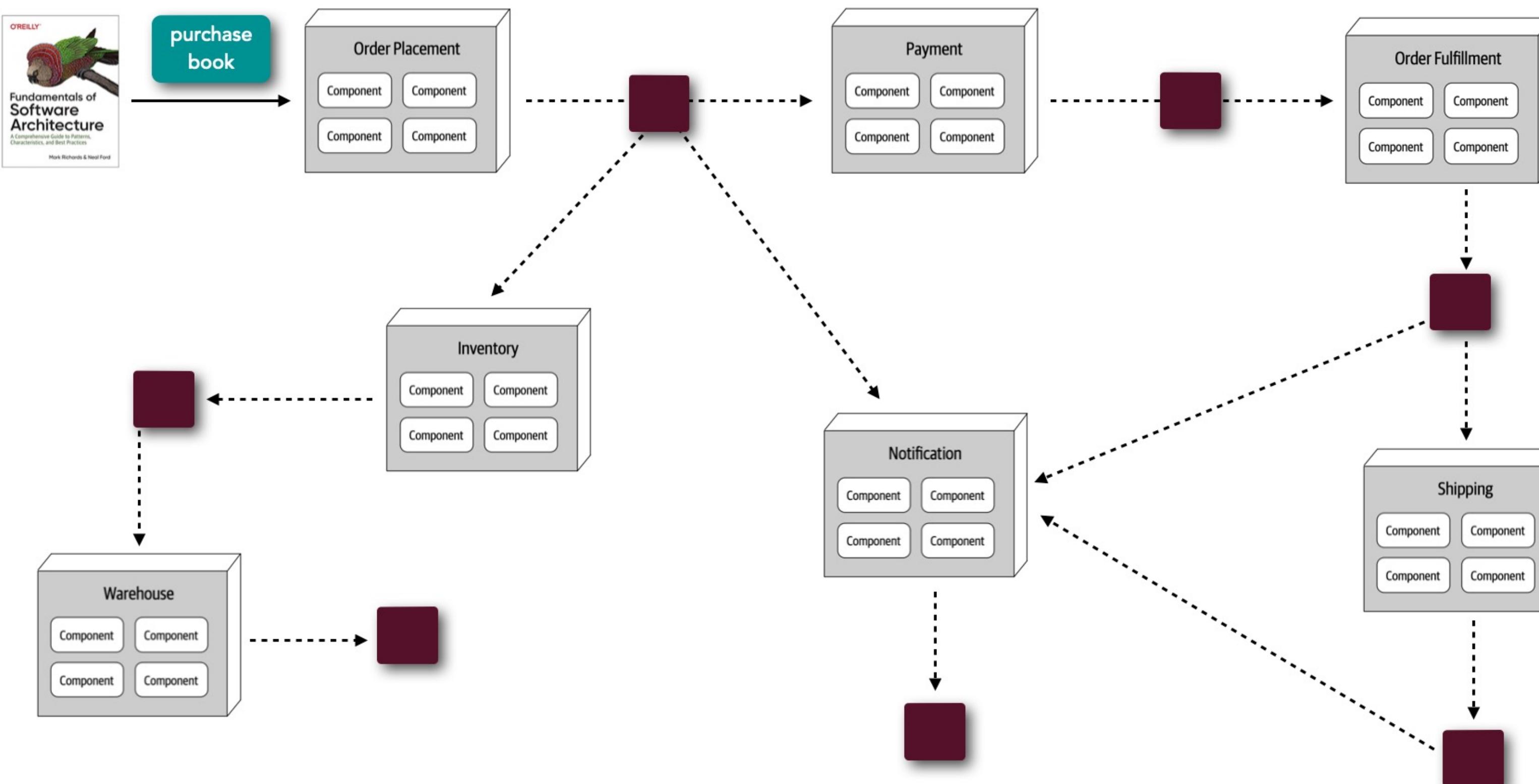


# event-driven architecture



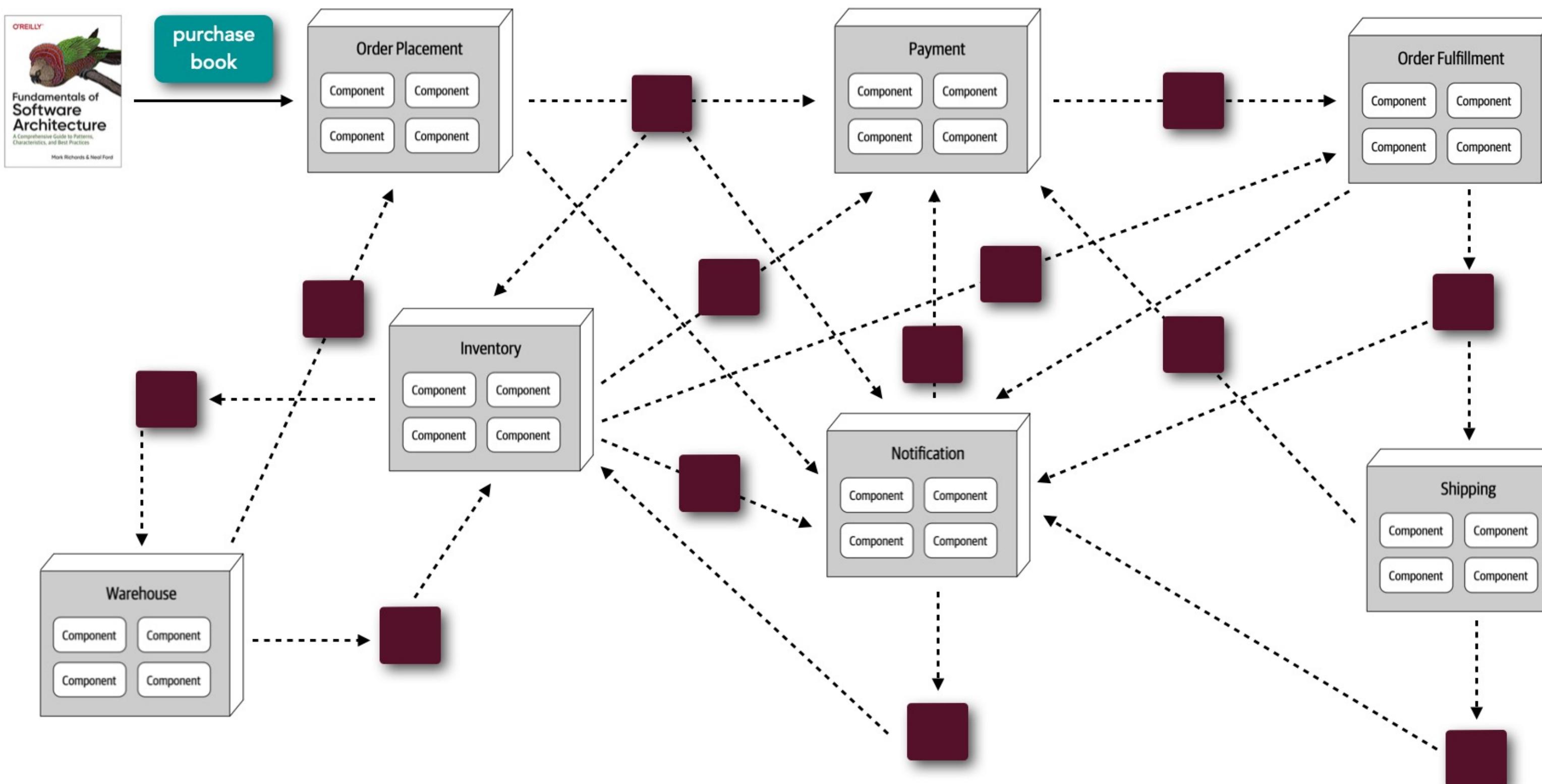
# event-driven architecture

*it's as simple as that! just sprinkle a bit of SQS here, a bit of SNS there...*



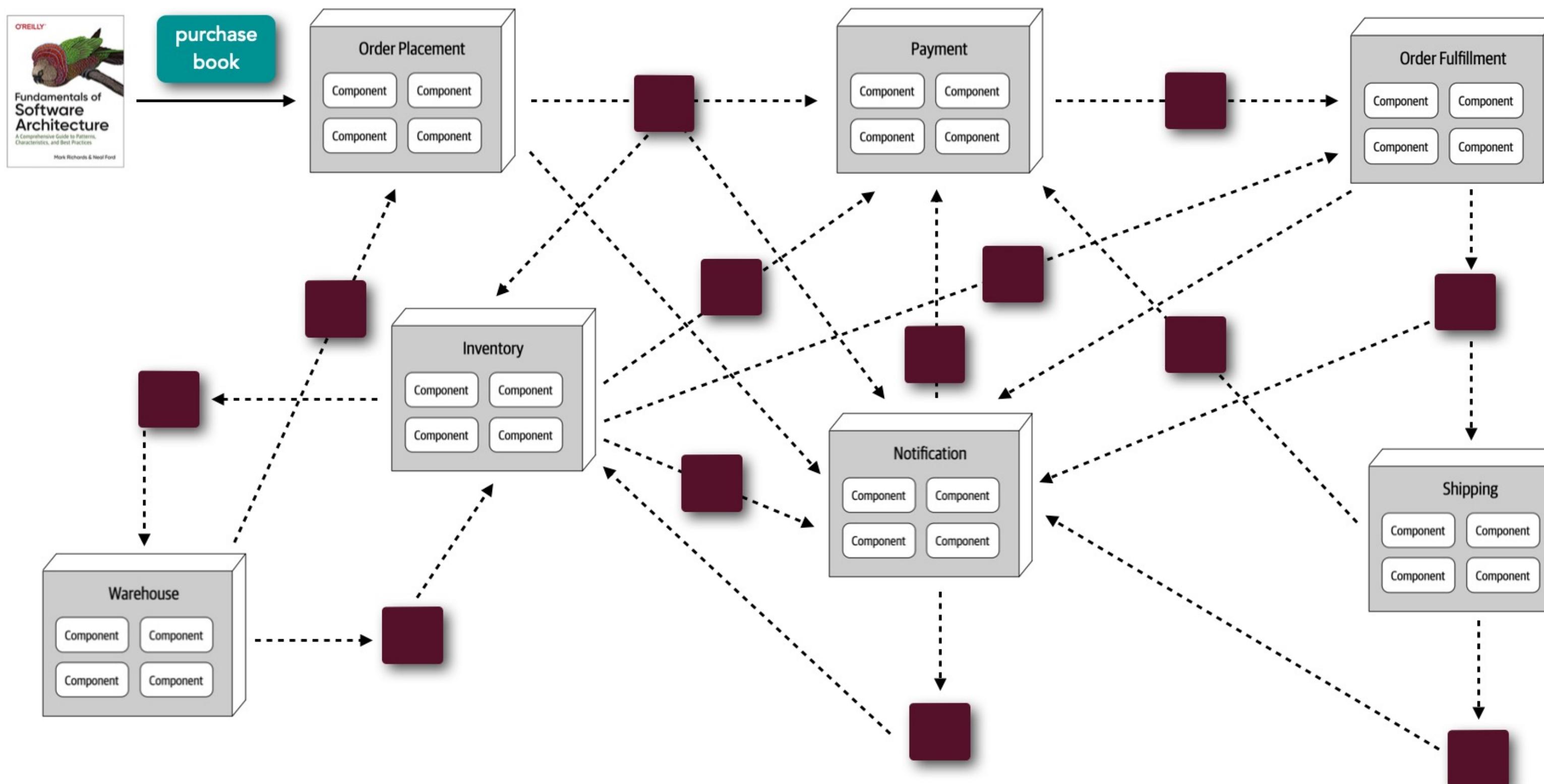
# event-driven architecture

*it's as simple as that! just sprinkle a bit of SQS here, a bit of SNS there...*



# event-driven architecture

*— it's as simple as that! just sprinkle a bit of SQS here, a bit of SNS there... —*



# event-driven architecture

*advantages*

*disadvantages*



# event-driven architecture

## *advantages*

- ✓ decoupling
- ✓ scalability
- ✓ fault tolerance
- ✓ responsiveness
- ✓ overall agility



# event-driven architecture

## advantages

- ✓ decoupling
- ✓ scalability
- ✓ fault tolerance
- ✓ responsiveness
- ✓ overall agility

## disadvantages

- ✗ error handling
- ✗ workflow control
- ✗ event timing
- ✗ contract coupling
- ✗ scenario testing



# event-driven architecture

## *advantages*

- ✓ decoupling
- ✓ scalability
- ✓ fault tolerance
- ✓ responsiveness
- ✓ overall agility



## *disadvantages*

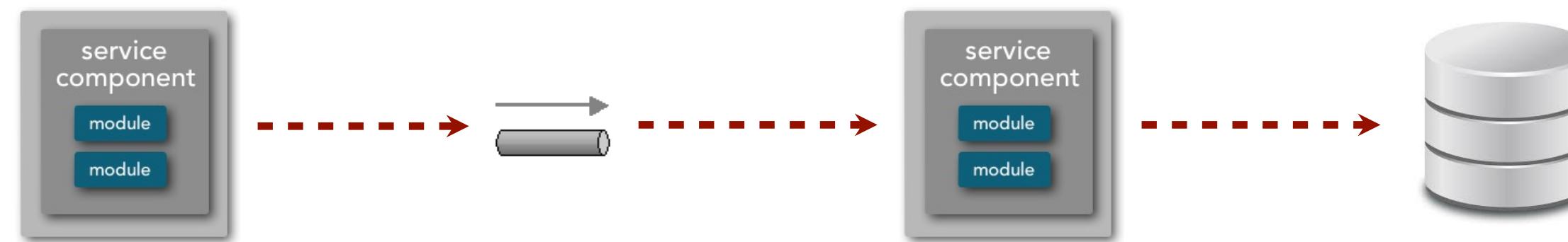
- ✗ error handling
- ✗ workflow control
- ✗ event timing
- ✗ contract coupling
- ✗ scenario testing

*increased complexity!*

# agenda

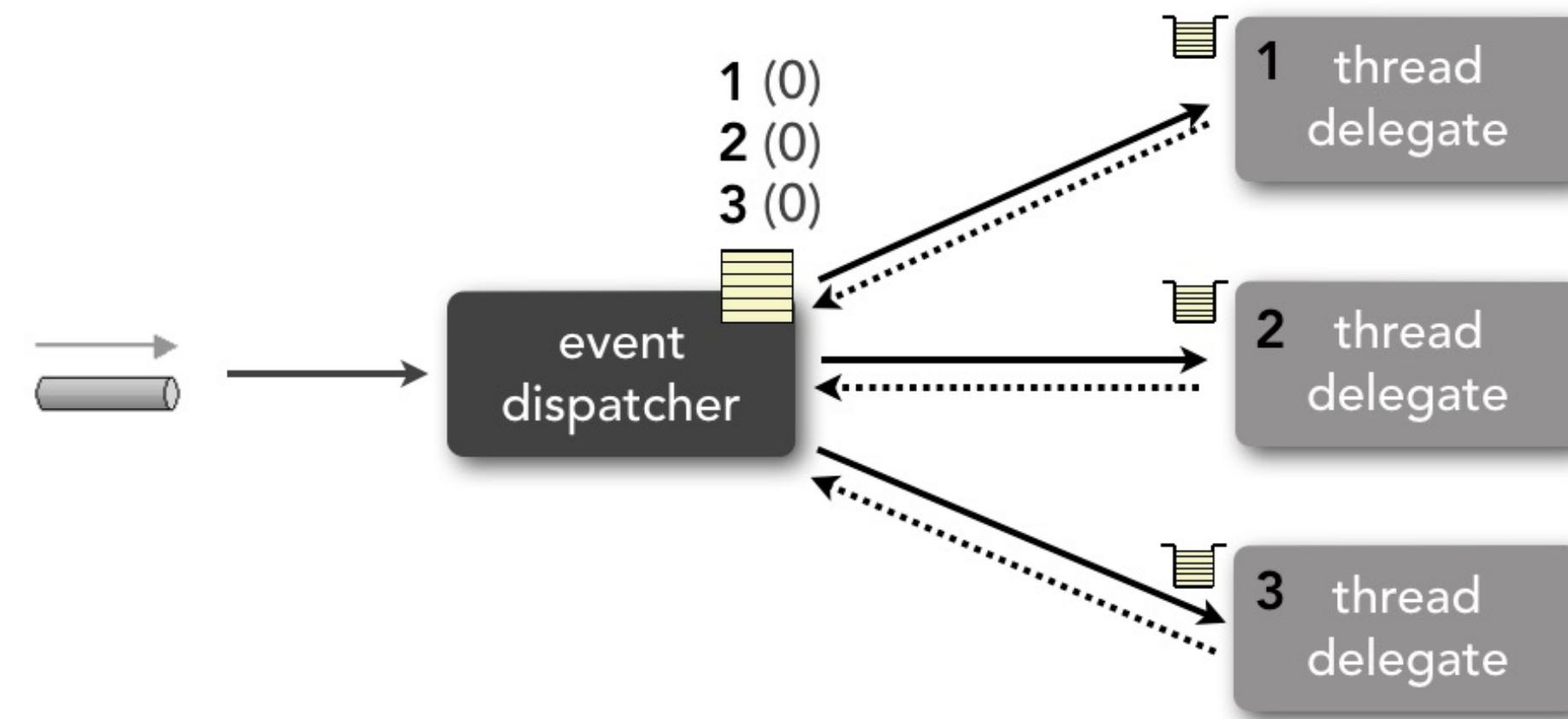
# agenda

*“how do I guarantee that no messages are lost when passing data from one service to another?”*



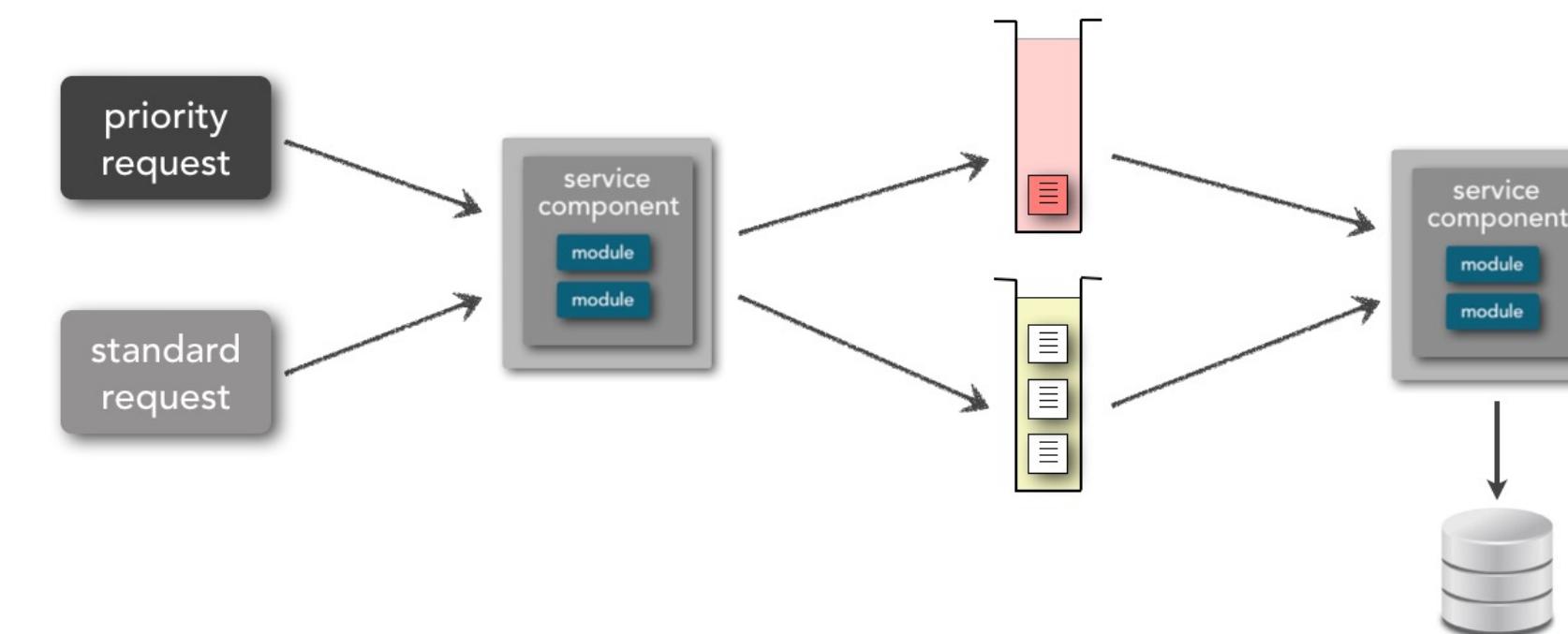
# agenda

*“how do I increase performance and throughput while still maintaining message processing order?”*



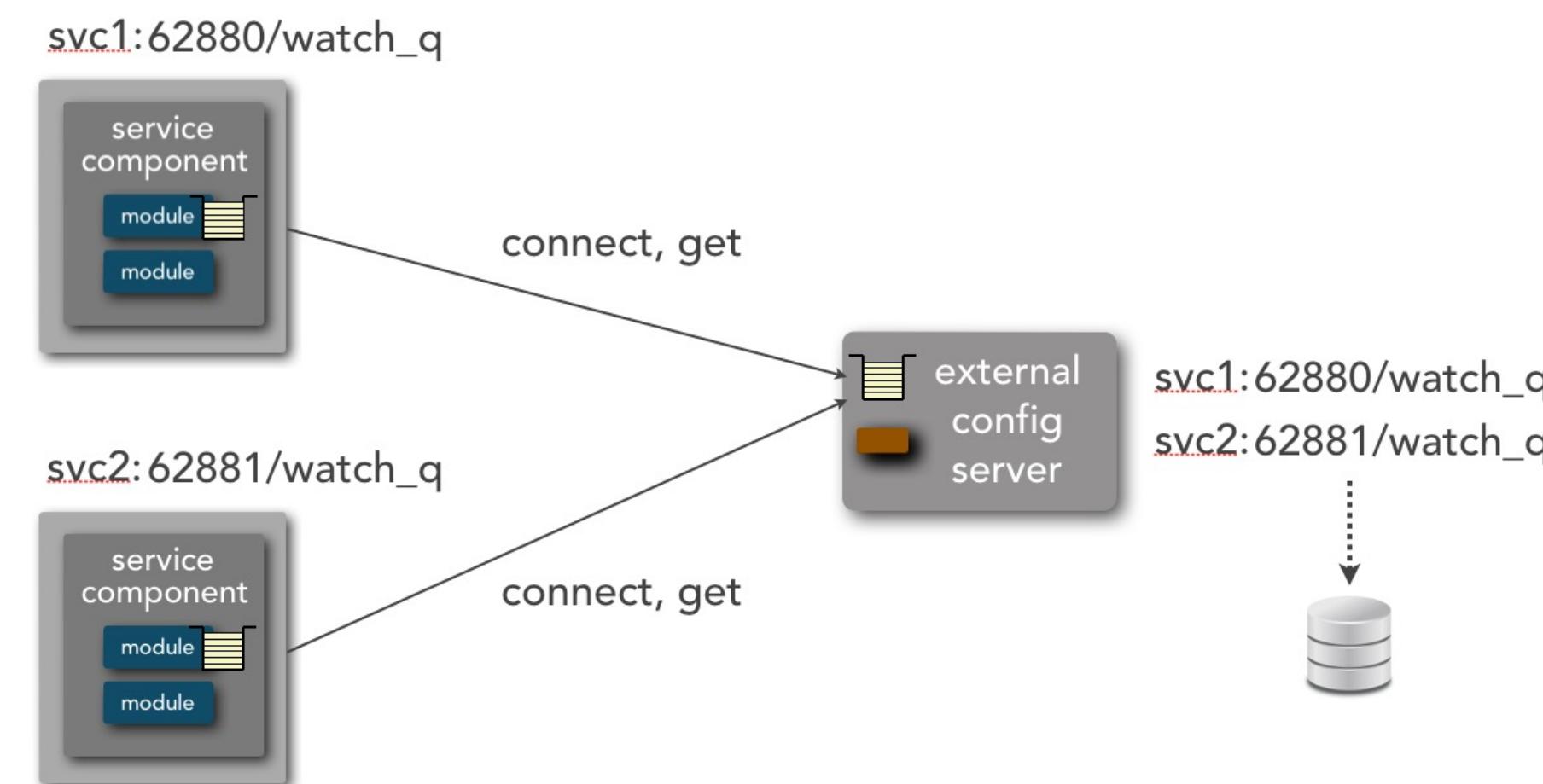
# agenda

*“how do I give some events a higher priority and still maintain responsiveness for all other events?”*



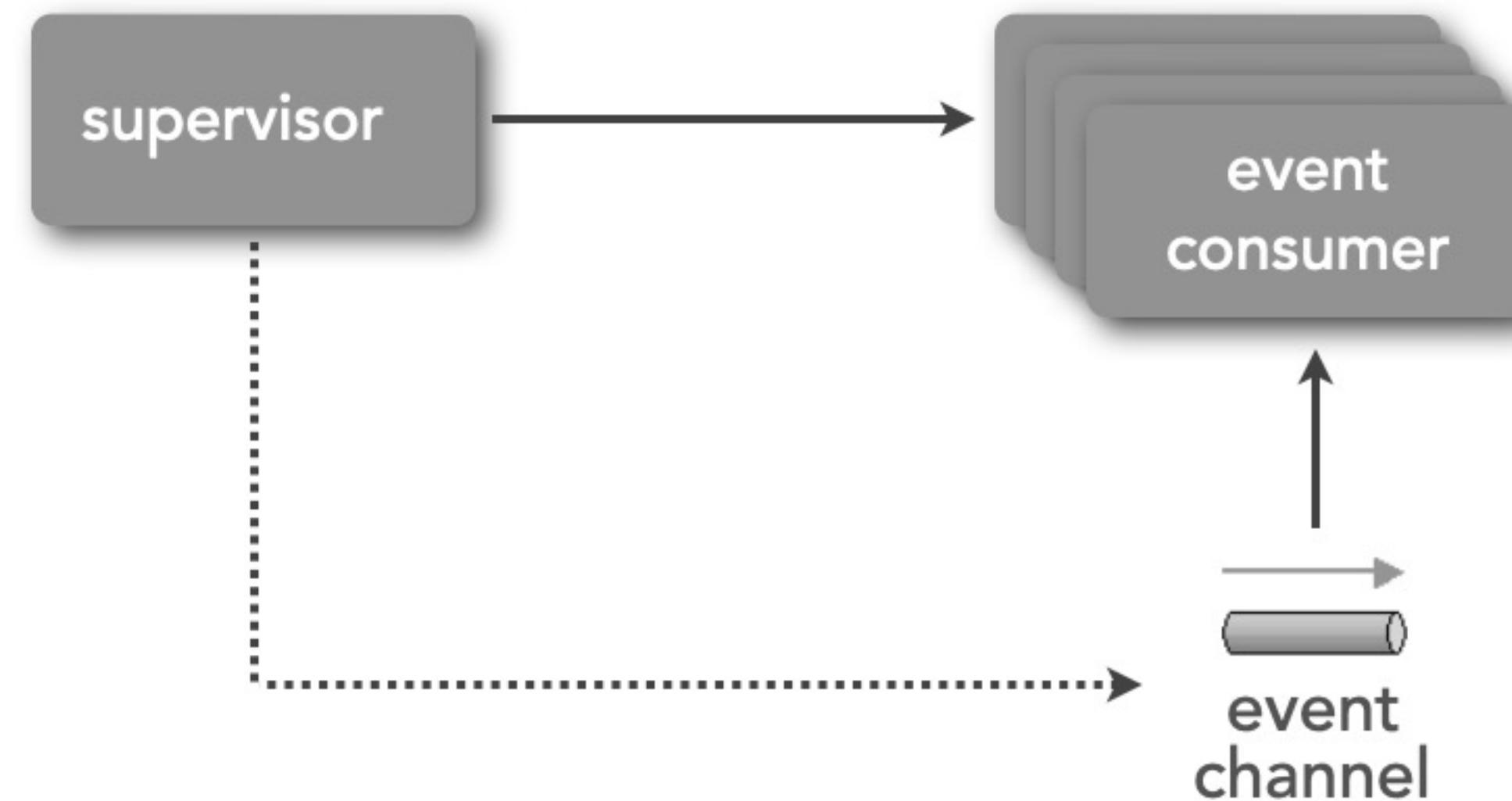
# agenda

*“how do I send notification events to services without maintaining a persistent connection to those services?”*



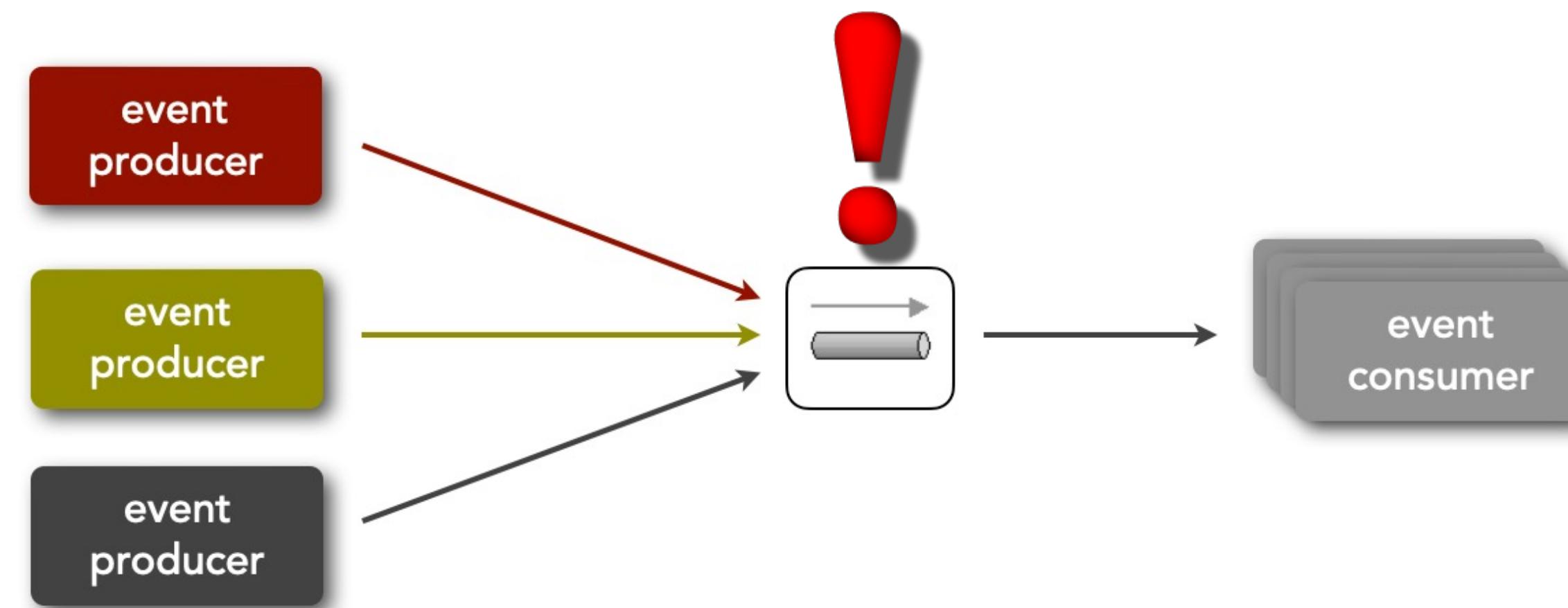
# agenda

*“how do I handle varying request load and still maintain a consistent response time?”*



# agenda

*“how do I increase the throughput and capacity of events through the system?”*



# the tradeoffs...



# the tradeoffs...

the good things



# the tradeoffs...



the bad things

# source code



<https://github.com/wmr513/event-driven-patterns>

<https://github.com/wmr513/reactive>

# Event Forwarding Pattern

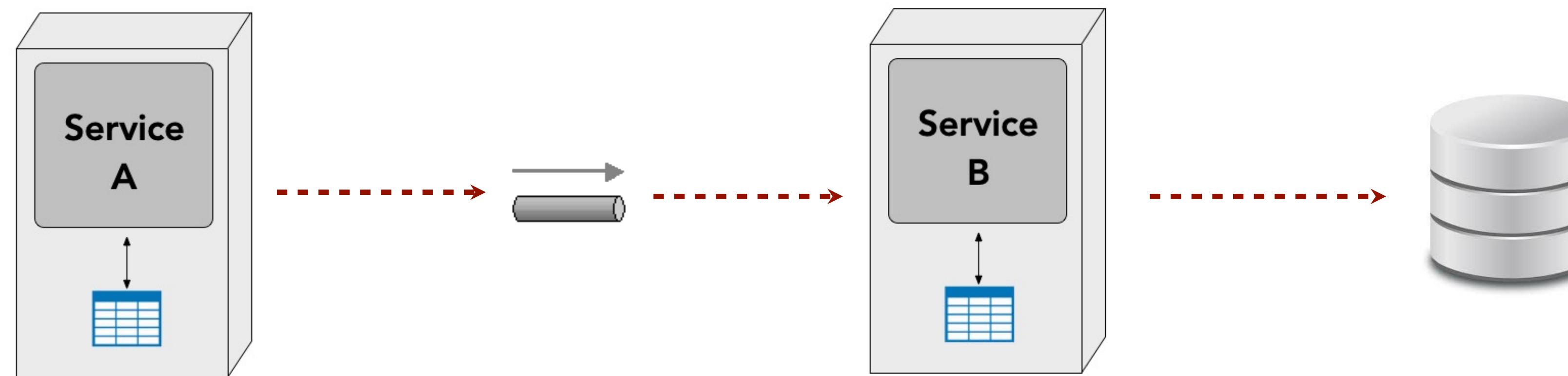
# event forwarding pattern



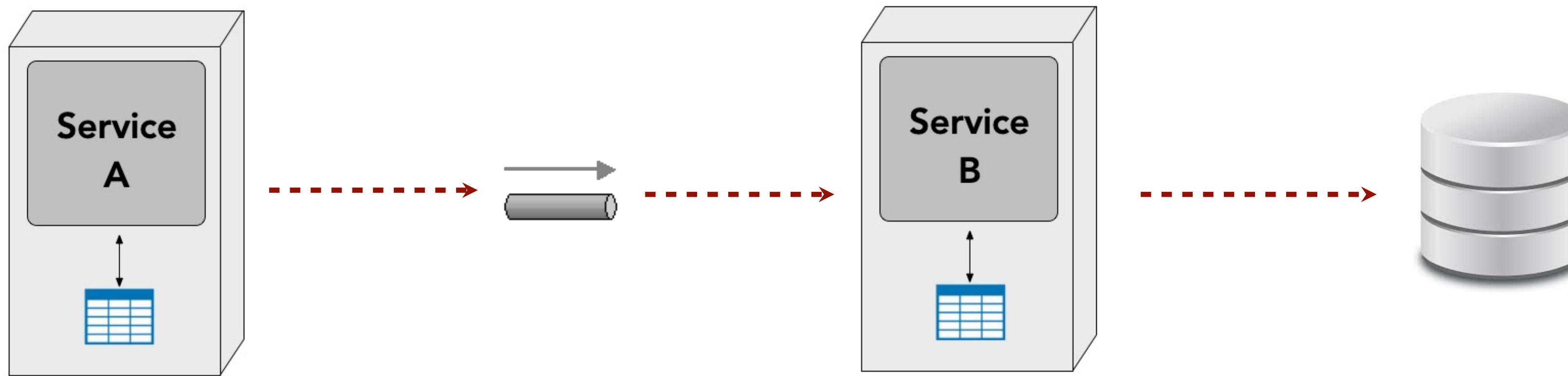
[https://github.com/wmr513/event-driven-patterns/  
tree/master/eventforwarding](https://github.com/wmr513/event-driven-patterns/tree/master/eventforwarding)

# event forwarding pattern

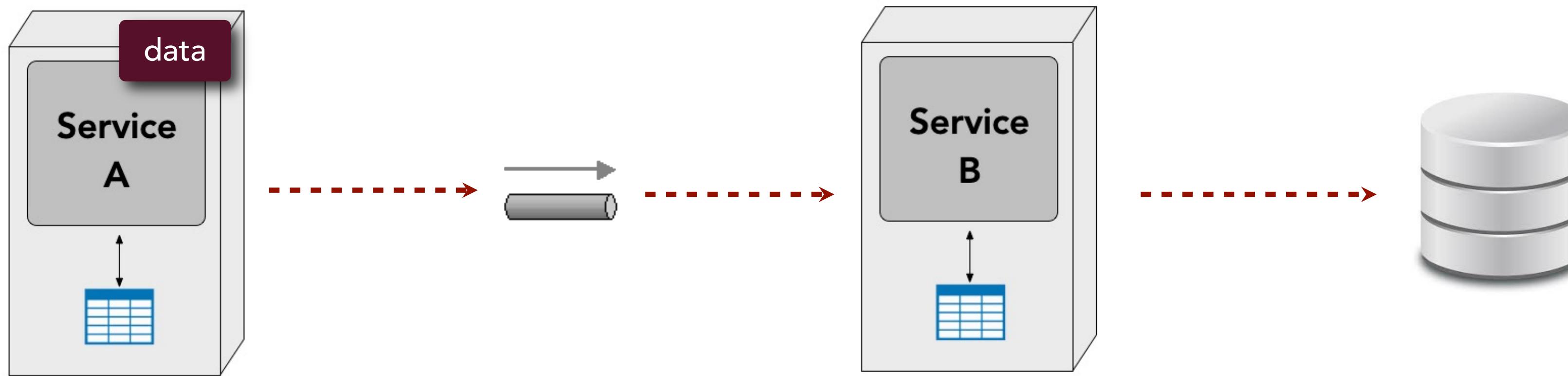
*“how do I guarantee that no messages are lost when passing data from one service to another?”*



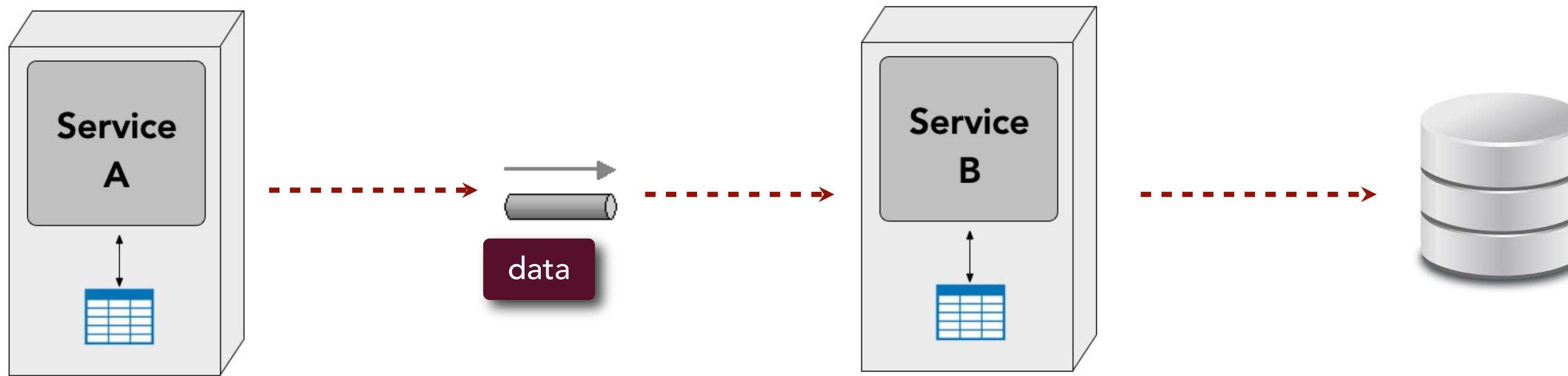
# event forwarding pattern



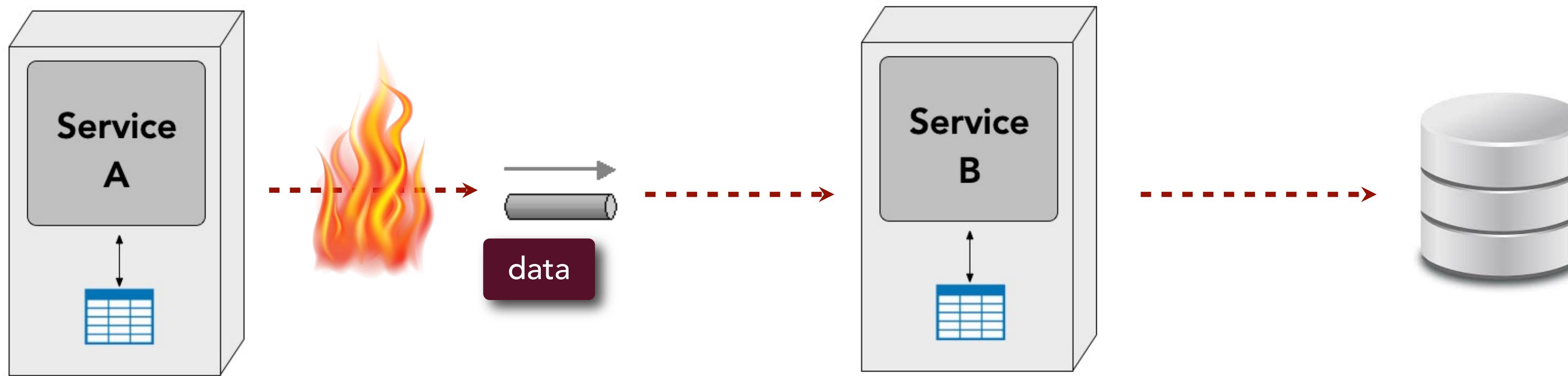
# event forwarding pattern



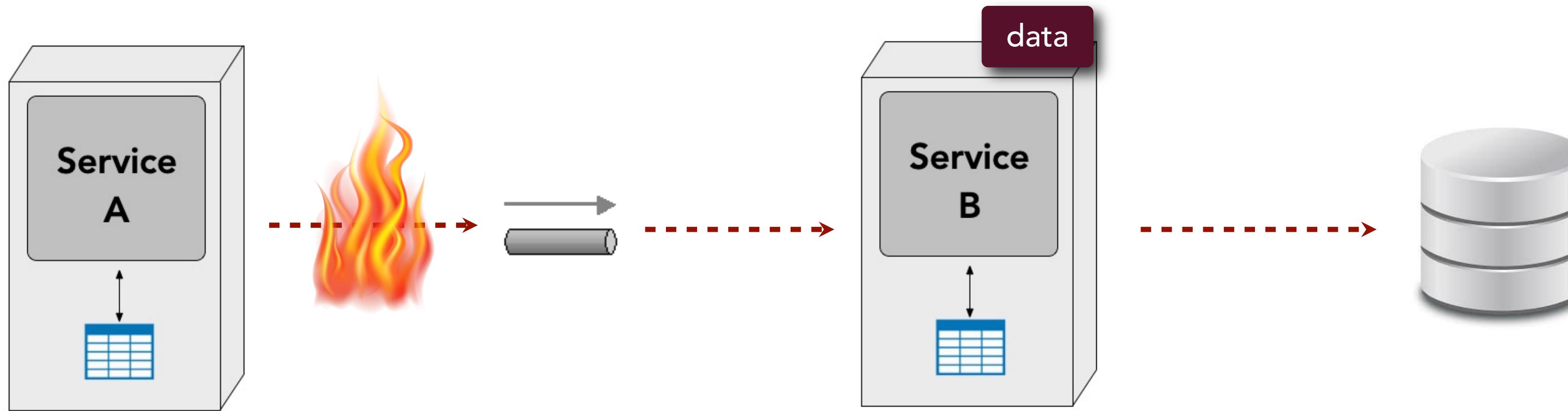
# event forwarding pattern



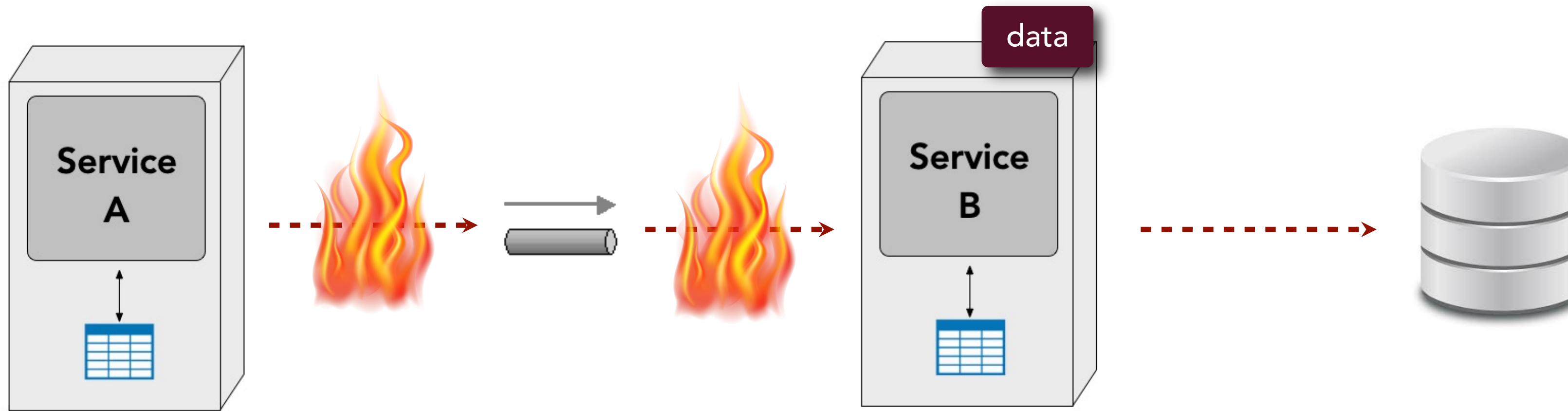
# event forwarding pattern



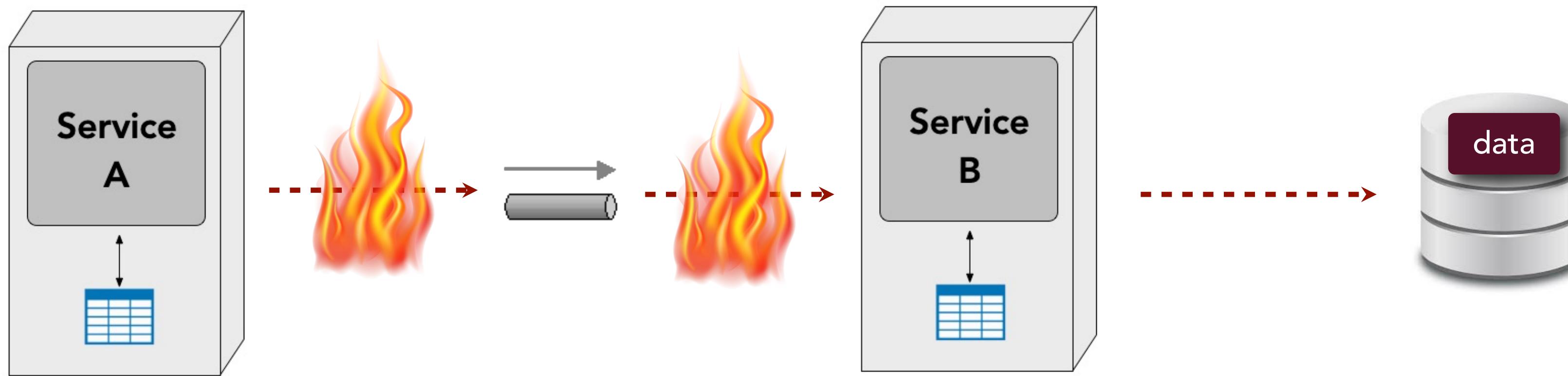
# event forwarding pattern



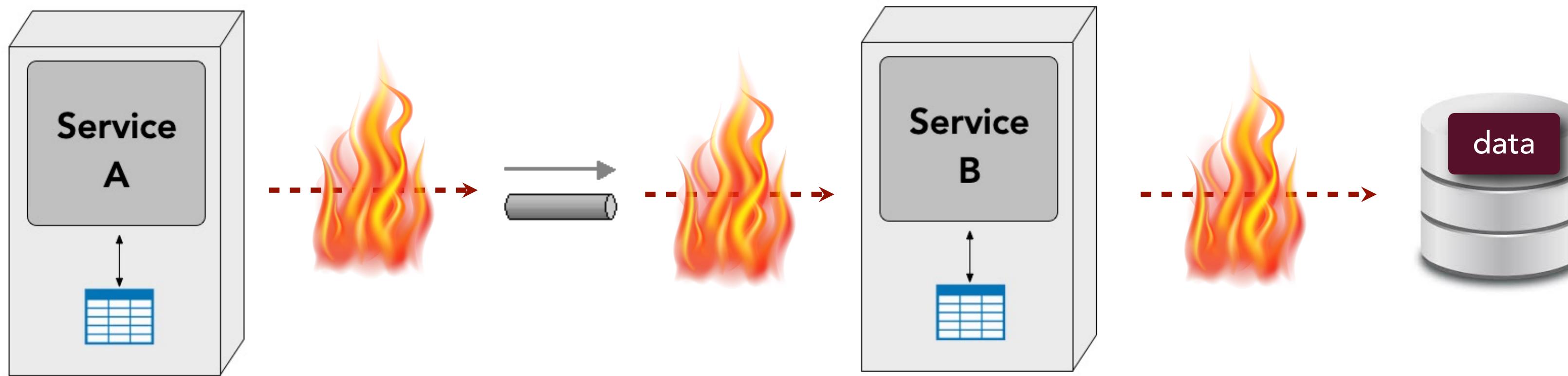
# event forwarding pattern



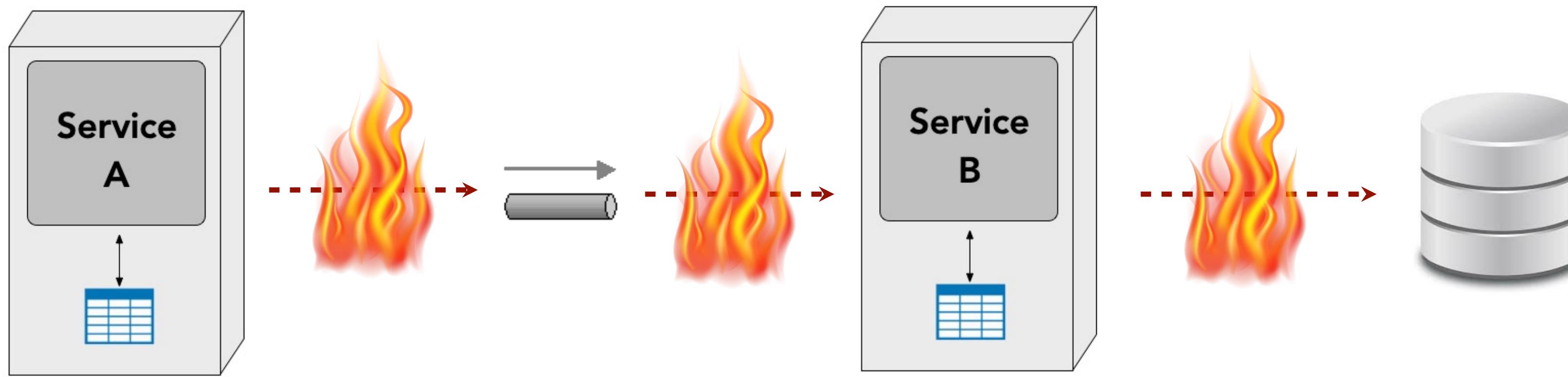
# event forwarding pattern



# event forwarding pattern



# event forwarding pattern

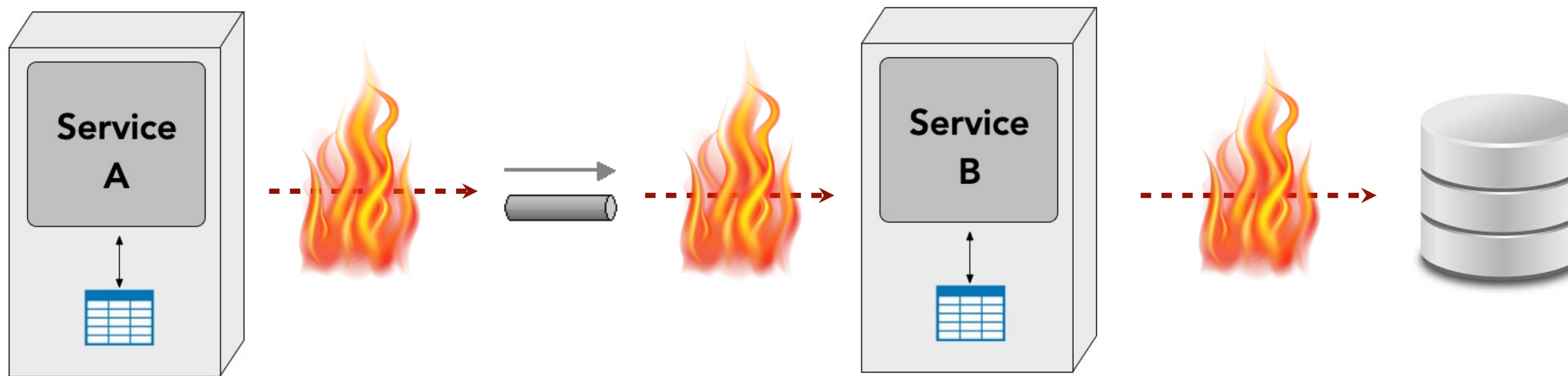


# event forwarding pattern

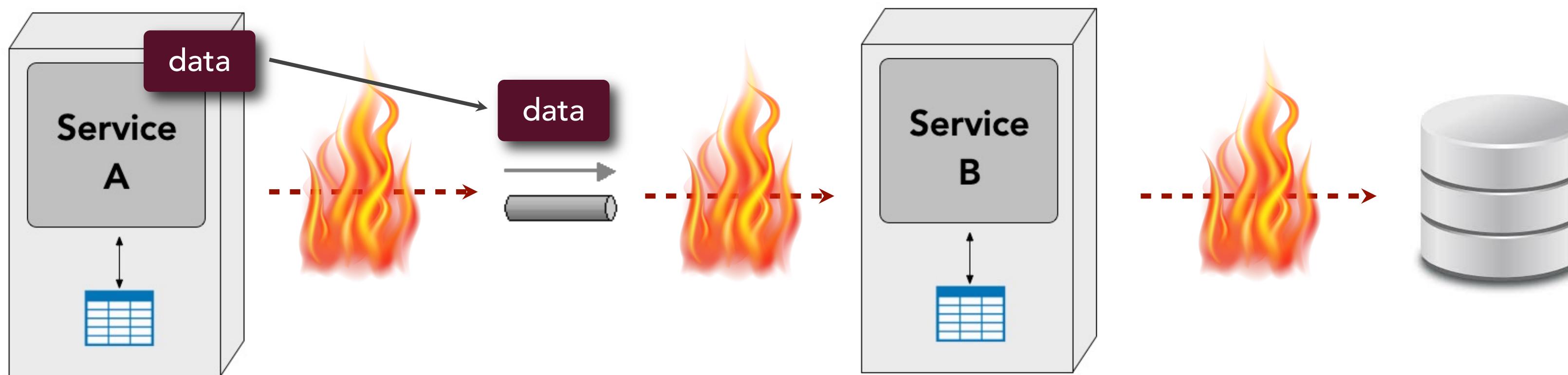


let's see the issue...

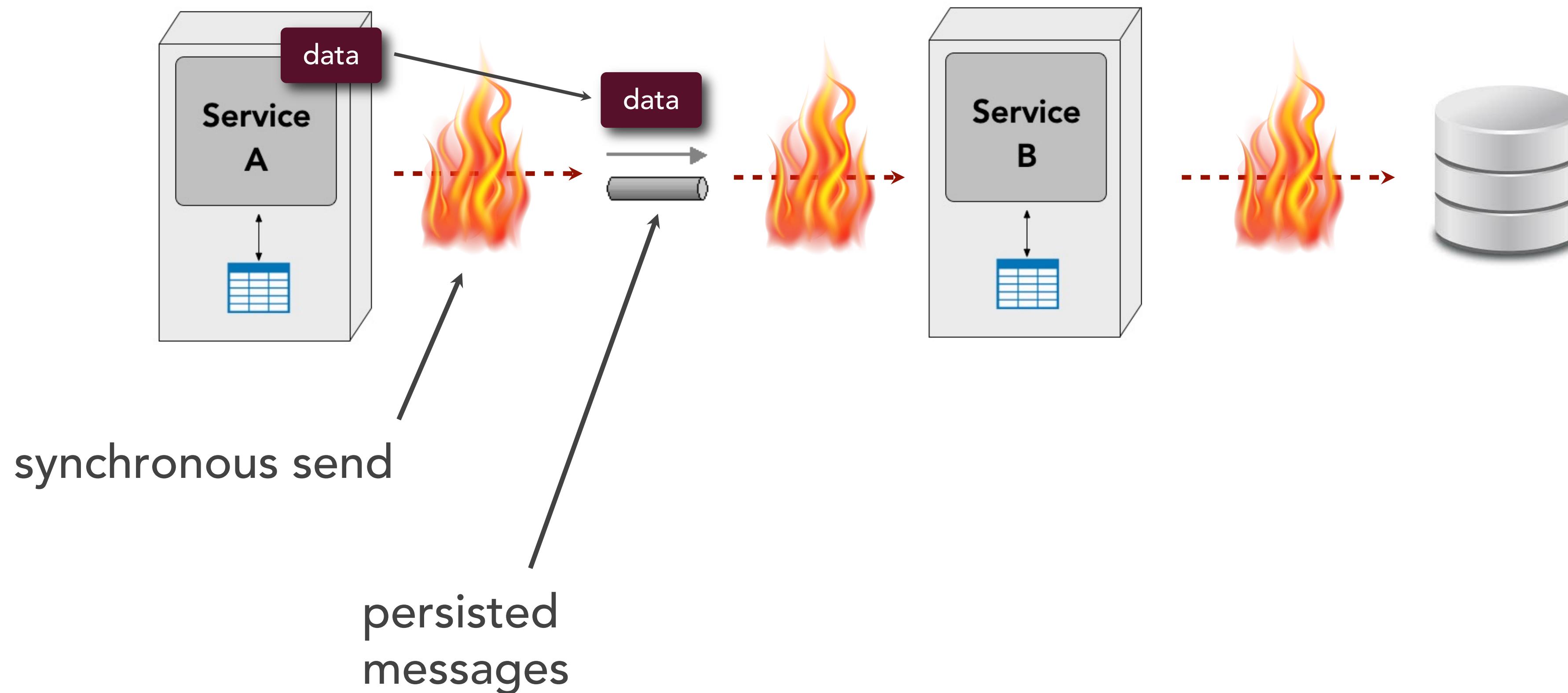
# event forwarding pattern



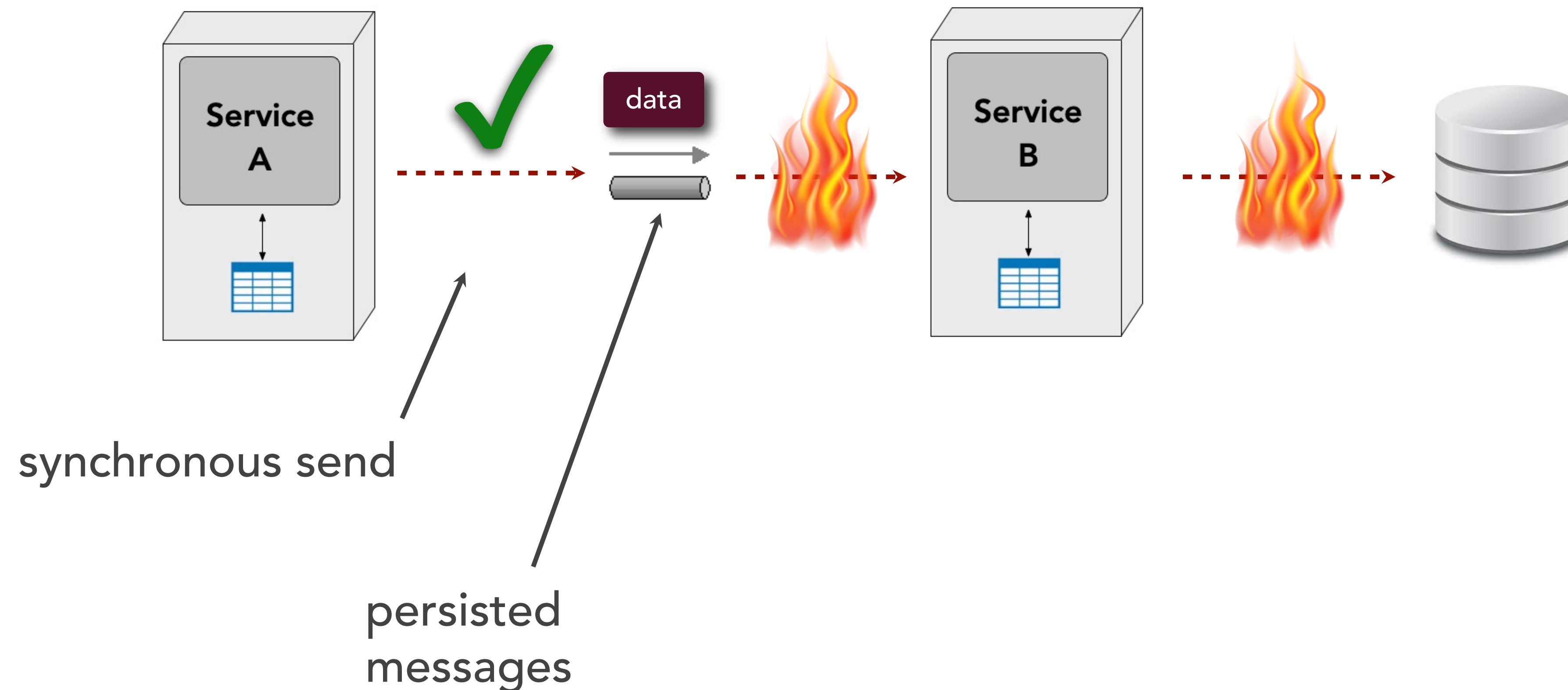
# event forwarding pattern



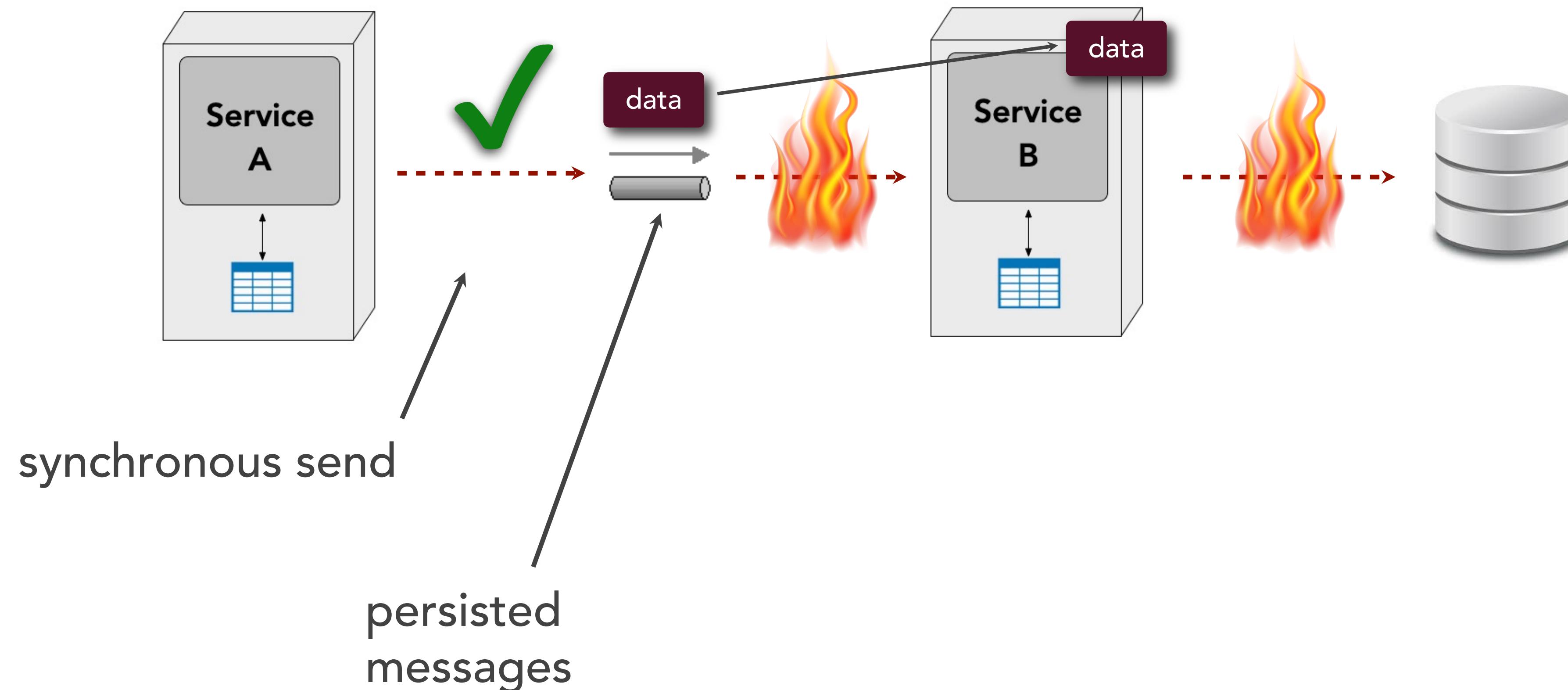
# event forwarding pattern



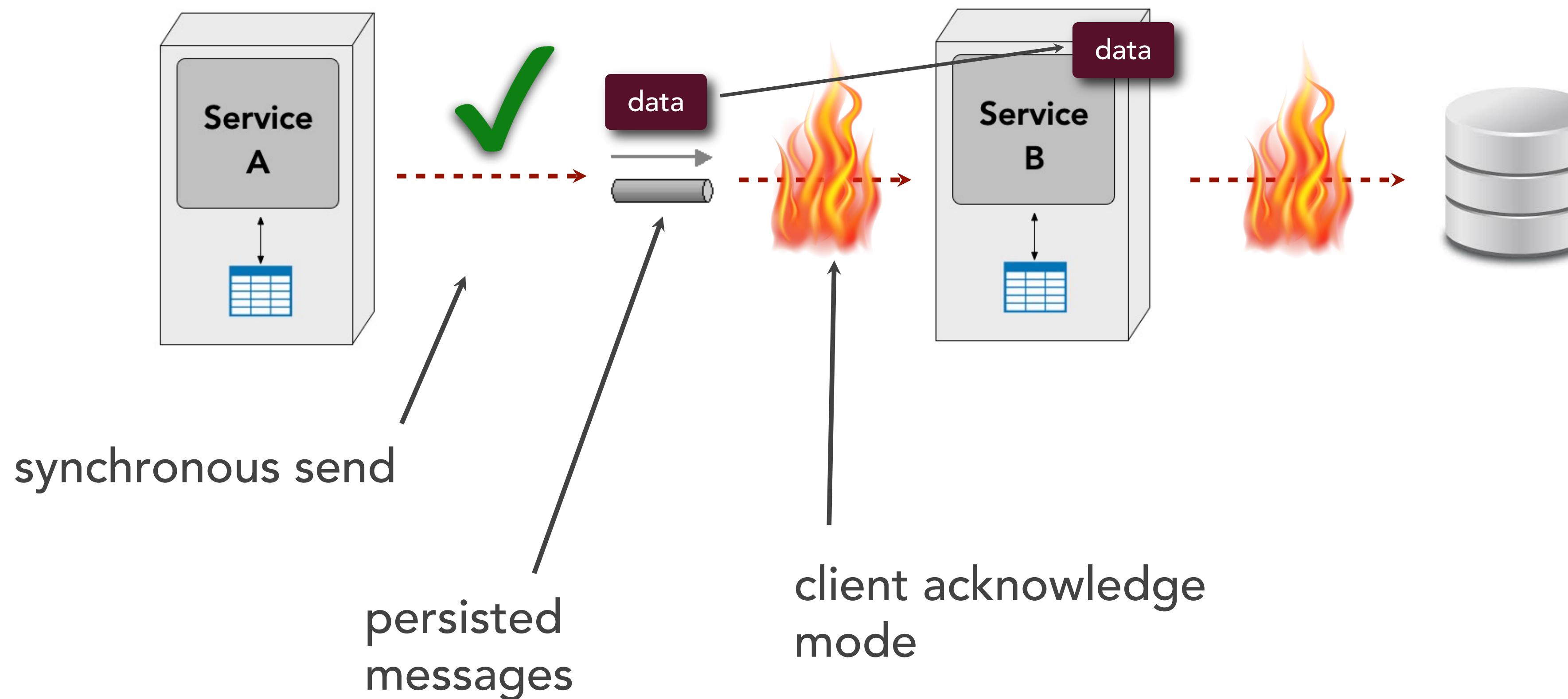
# event forwarding pattern



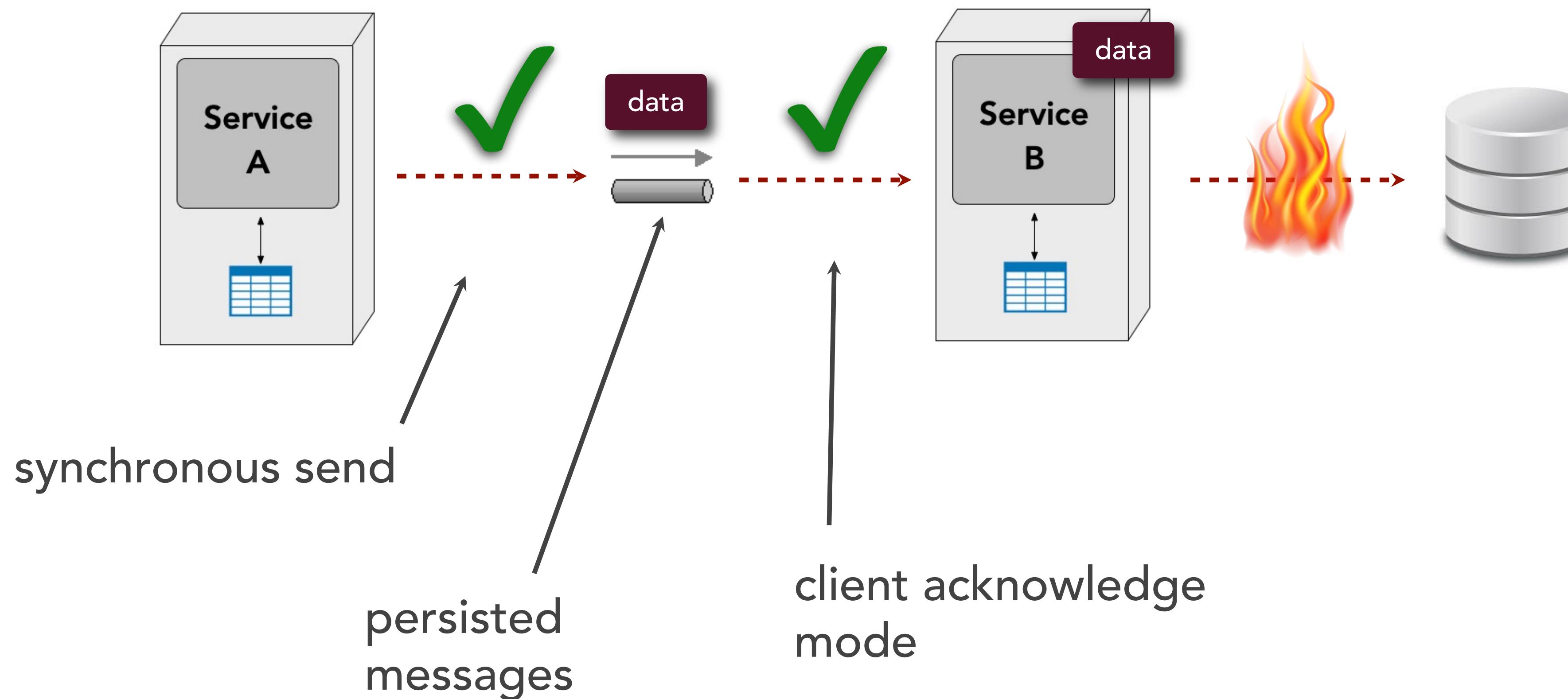
# event forwarding pattern



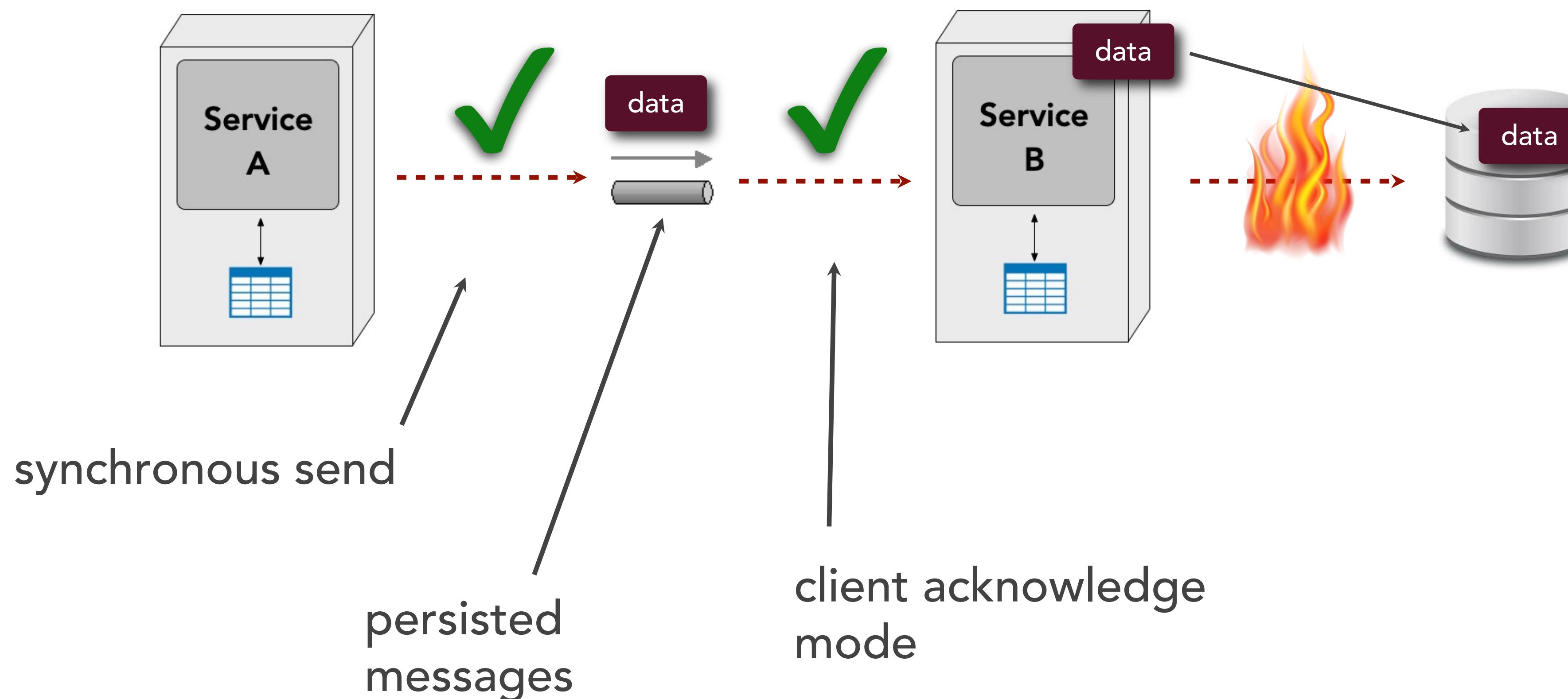
# event forwarding pattern



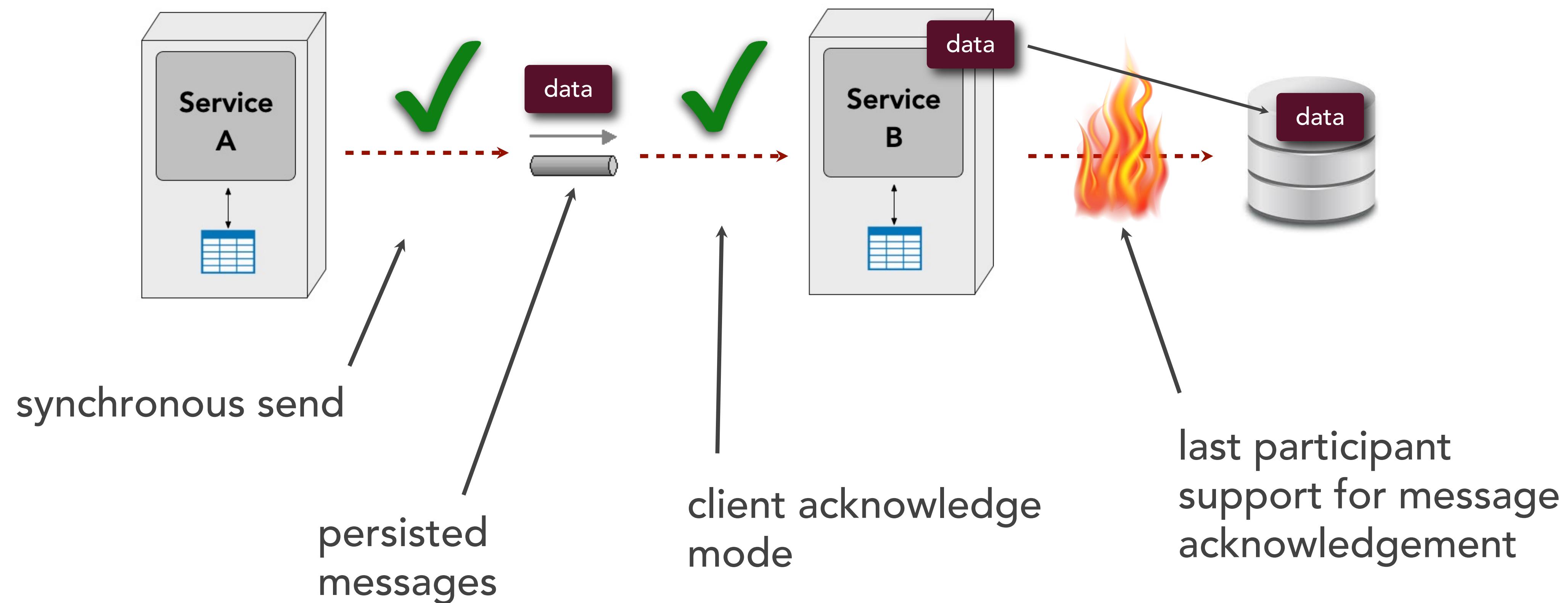
# event forwarding pattern



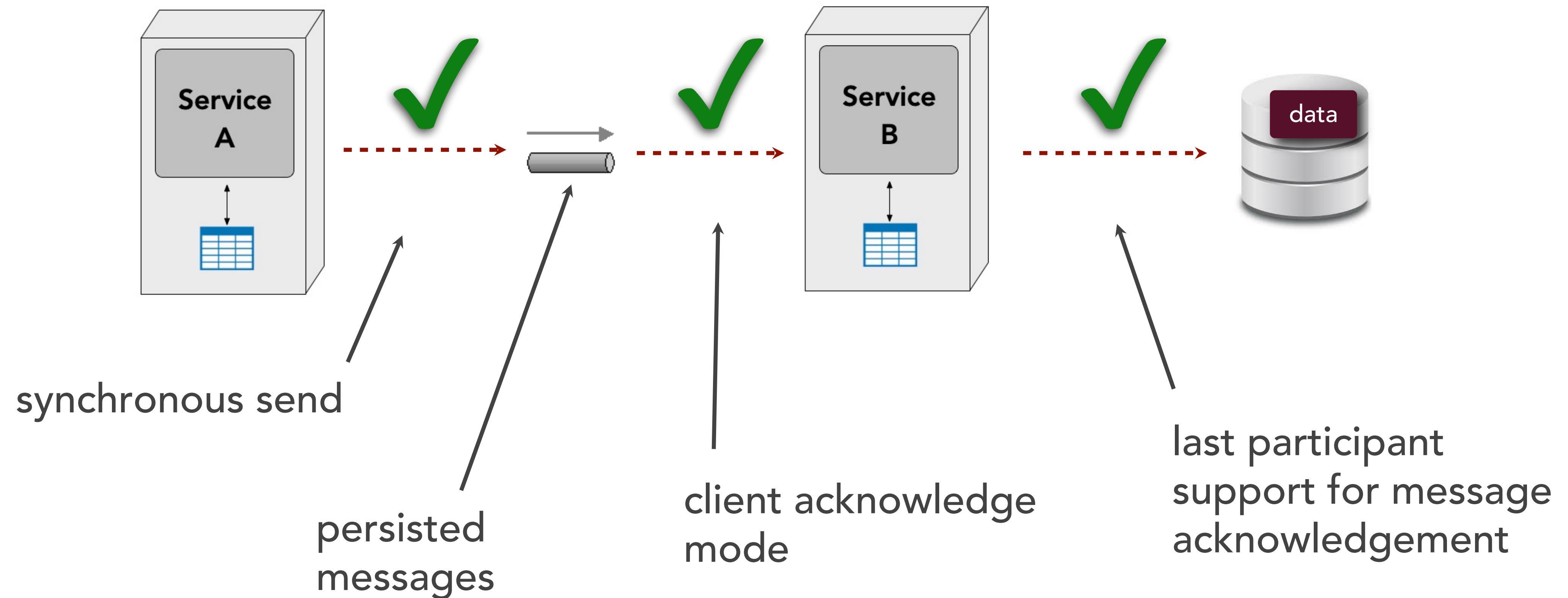
# event forwarding pattern



# event forwarding pattern



# event forwarding pattern

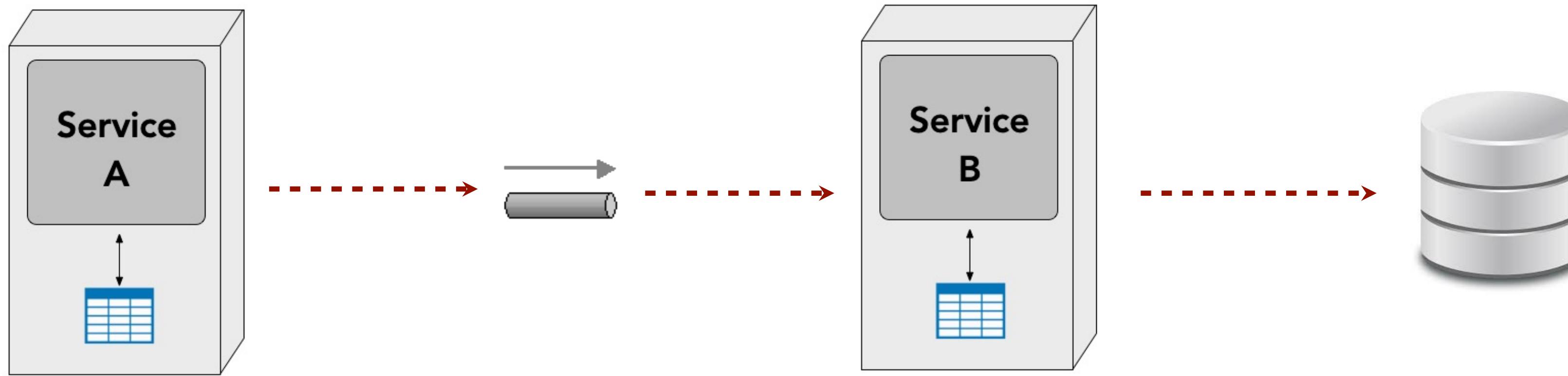


# event forwarding pattern

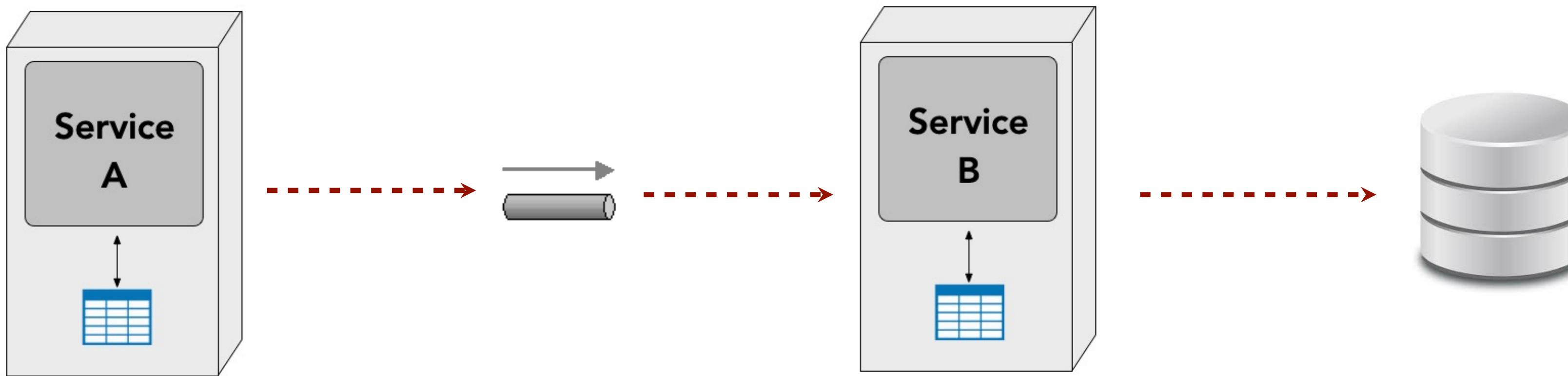


let's apply the pattern...

# event forwarding pattern



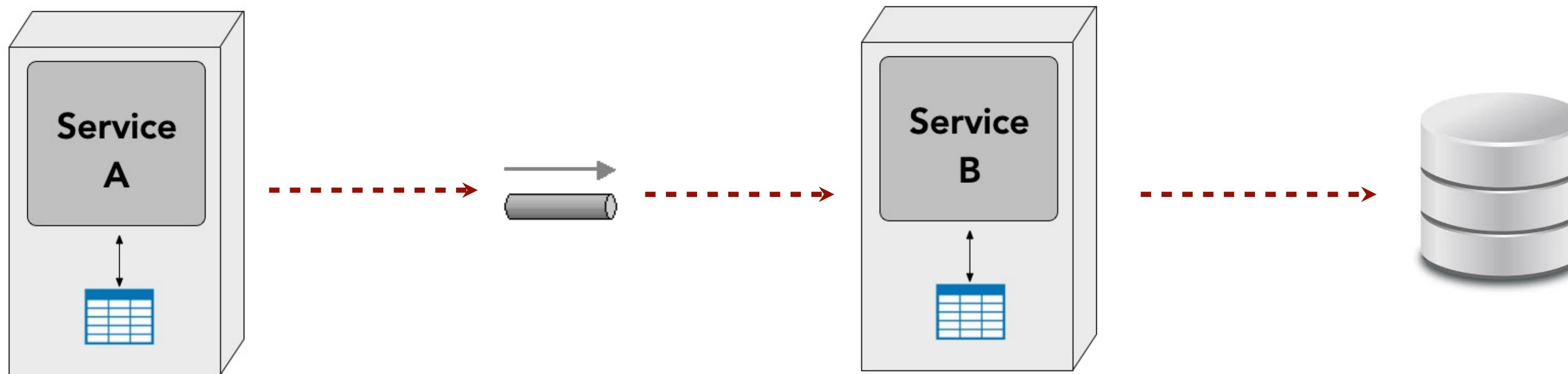
# event forwarding pattern



no message loss  
data integrity



# event forwarding pattern



no message loss  
data integrity



performance  
throughput  
possible duplicates

# Thread Delegate Pattern

# thread delegate pattern

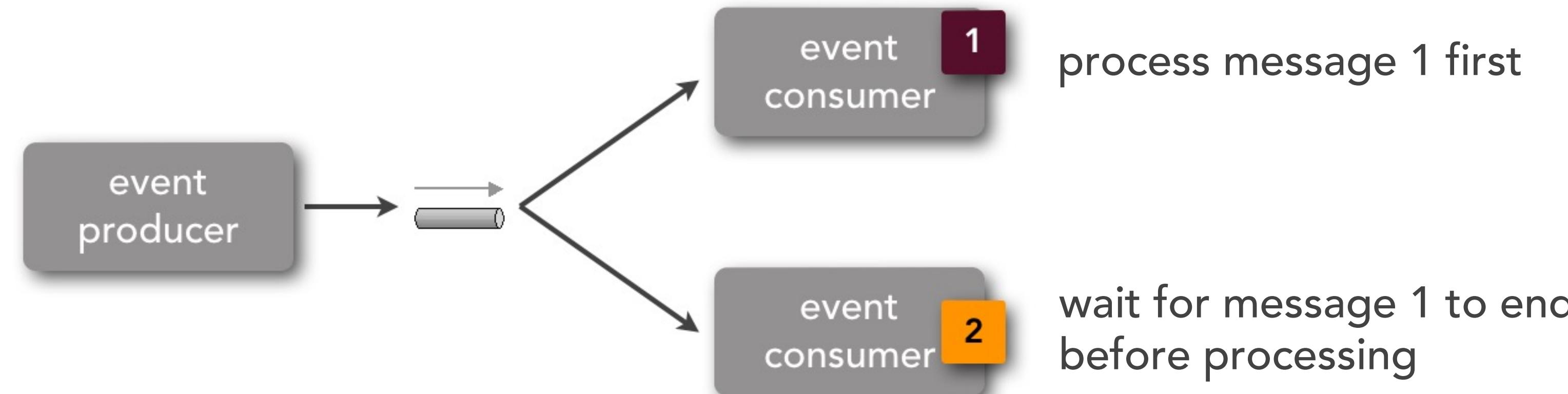
GitHub



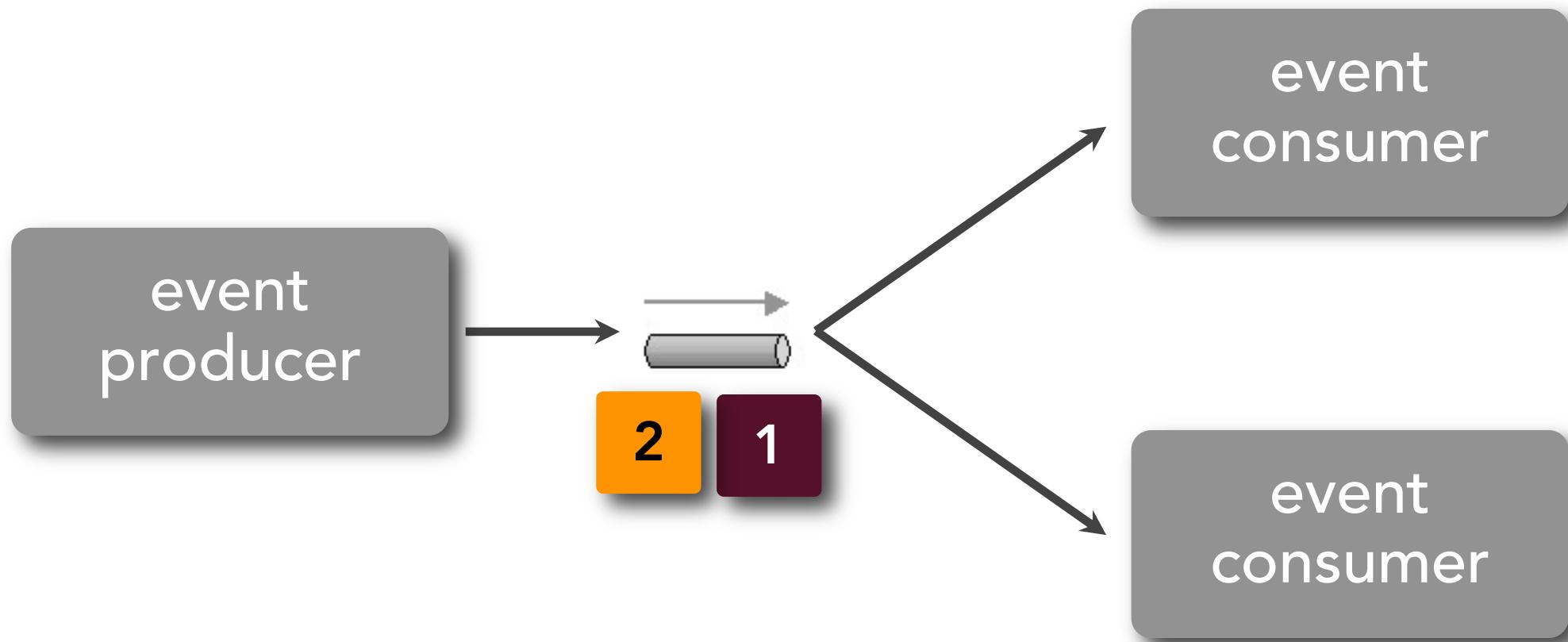
[https://github.com/wmr513/reactive/tree/master/  
threaddelegate](https://github.com/wmr513/reactive/tree/master/threaddelegate)

# thread delegate pattern

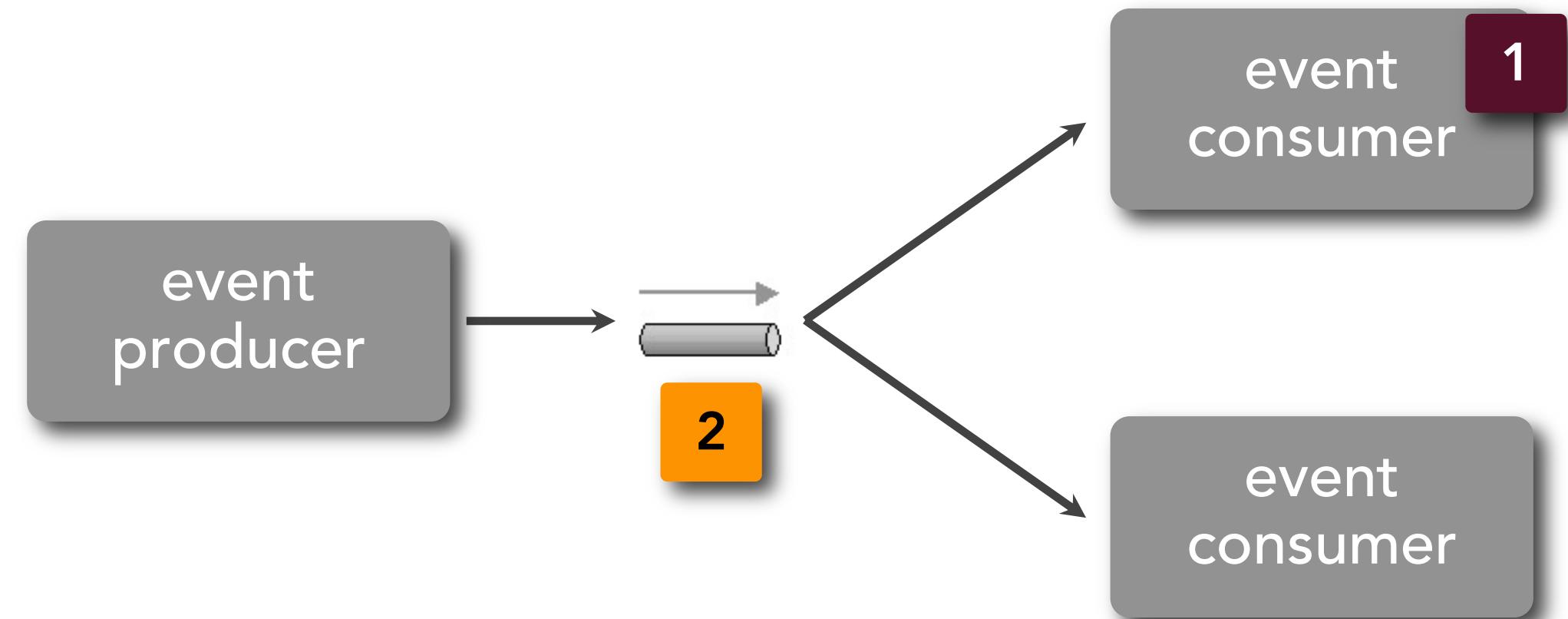
*“how can I increase performance and throughput while still maintaining message processing order?”*



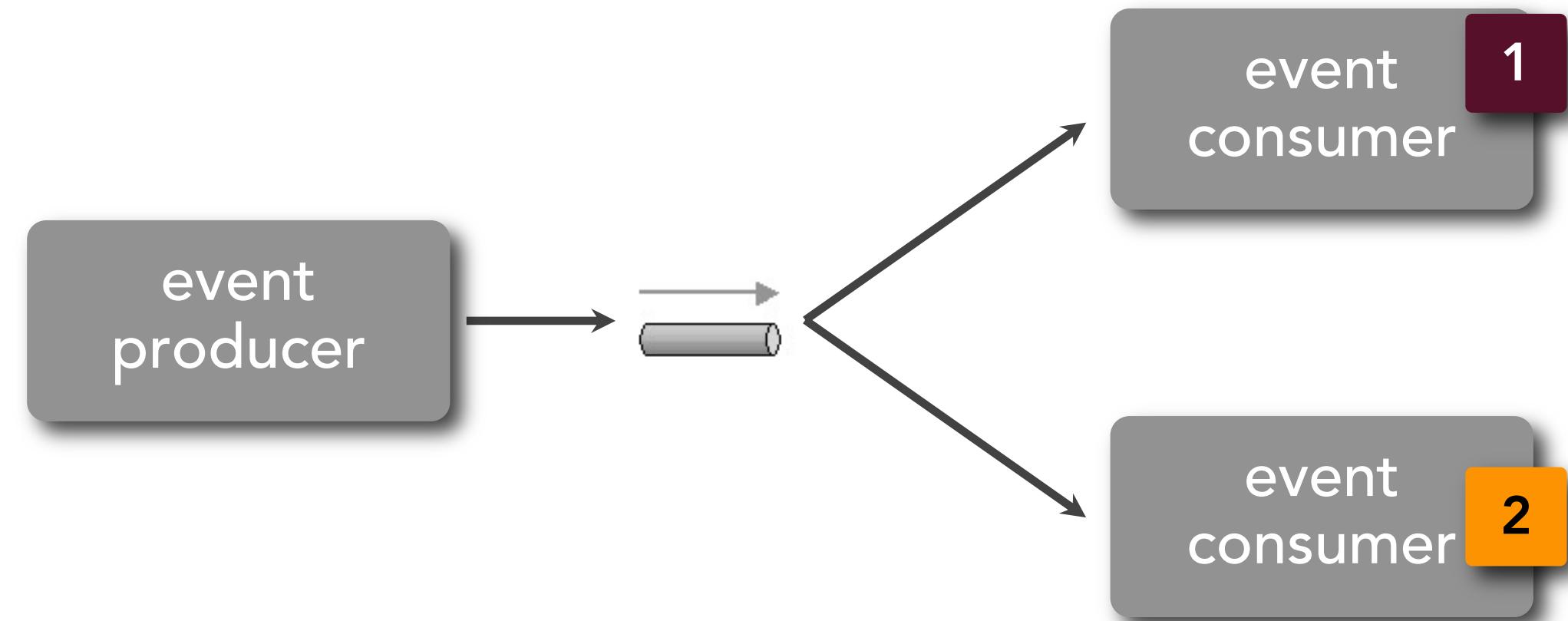
# thread delegate pattern



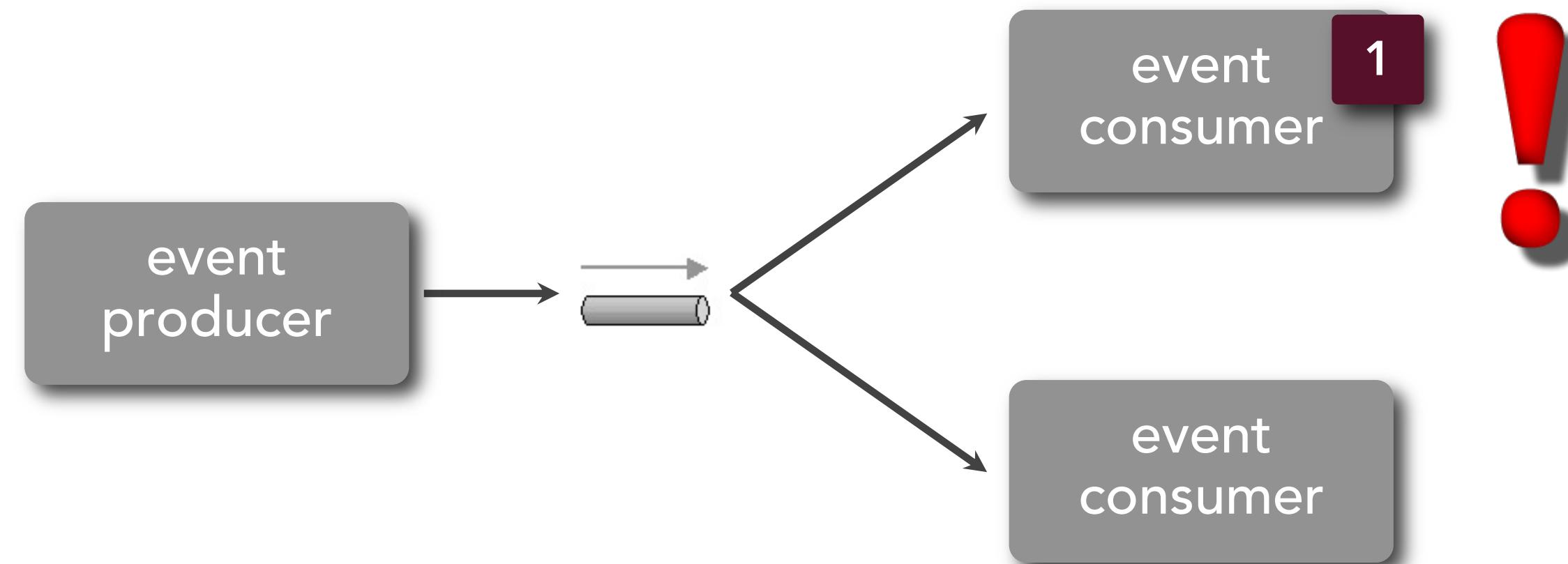
# thread delegate pattern



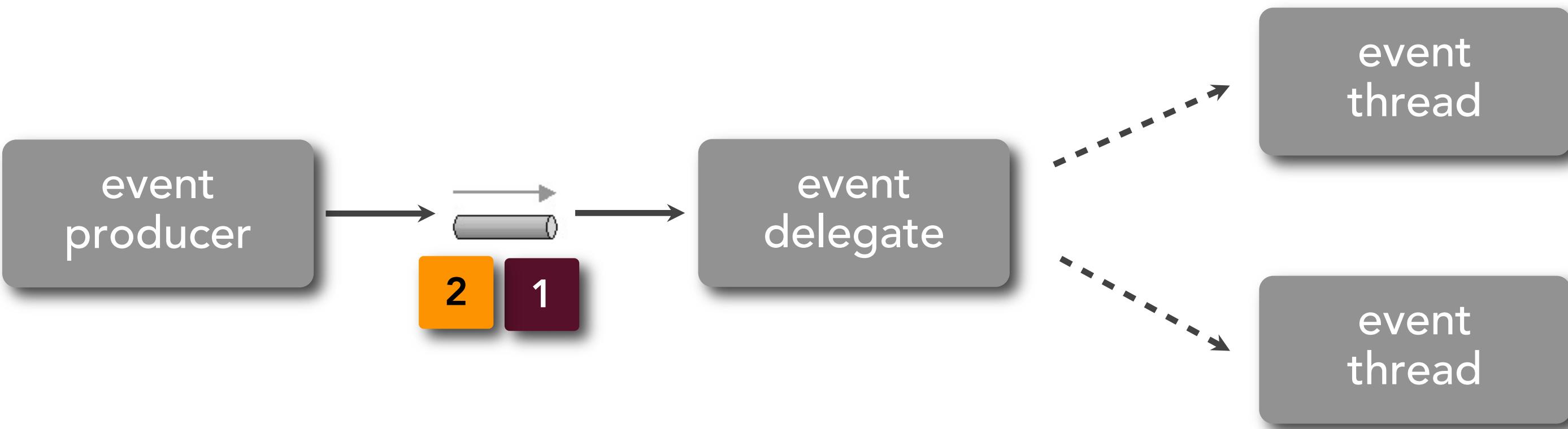
# thread delegate pattern



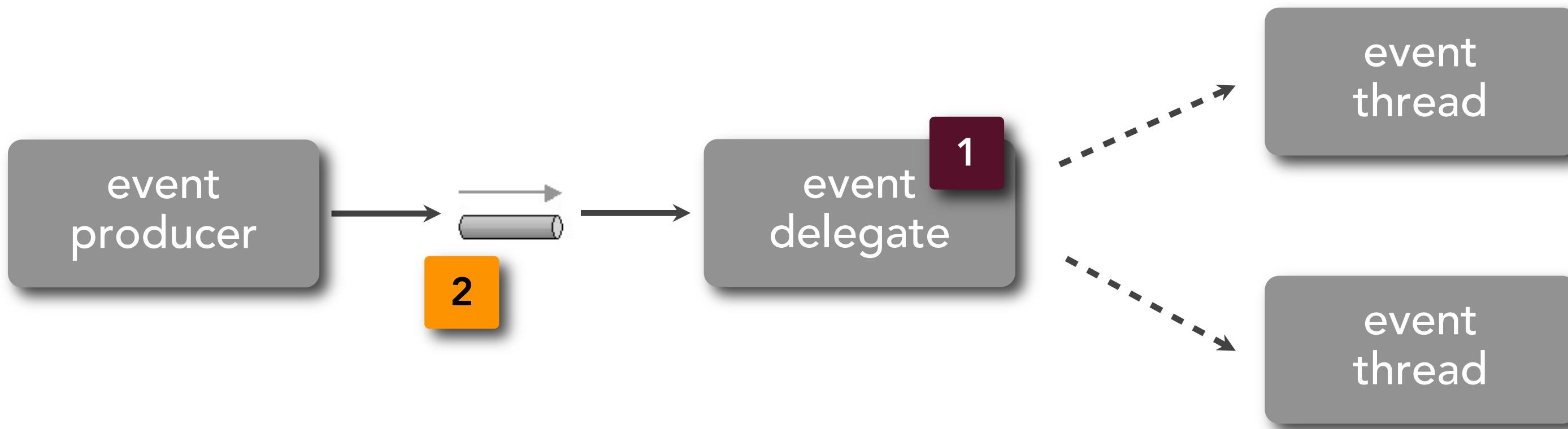
# thread delegate pattern



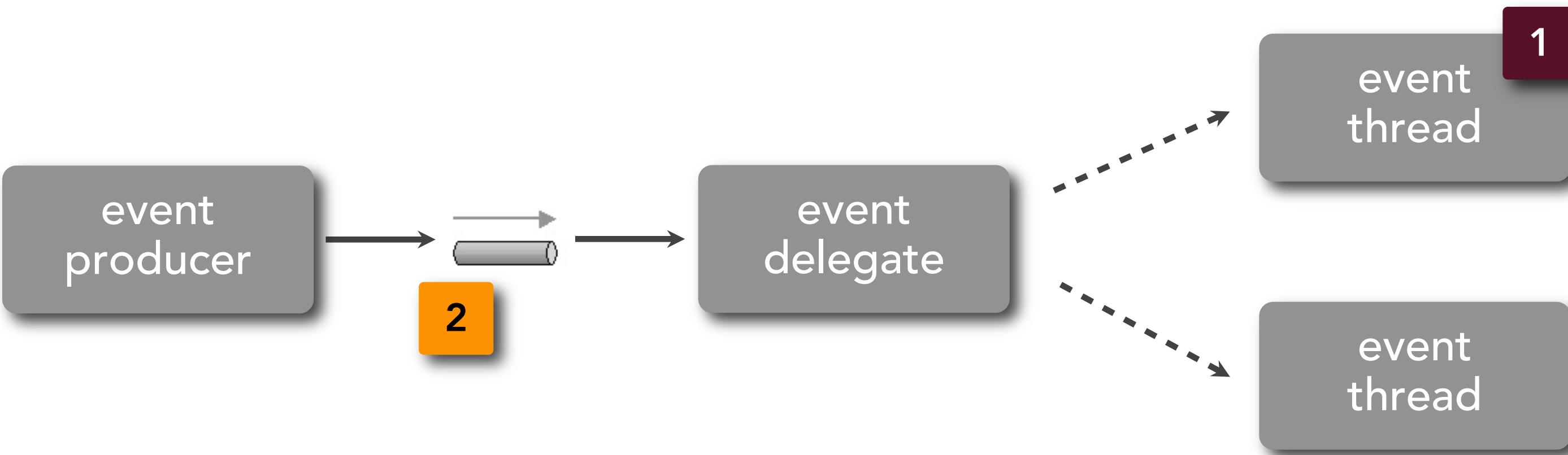
# thread delegate pattern



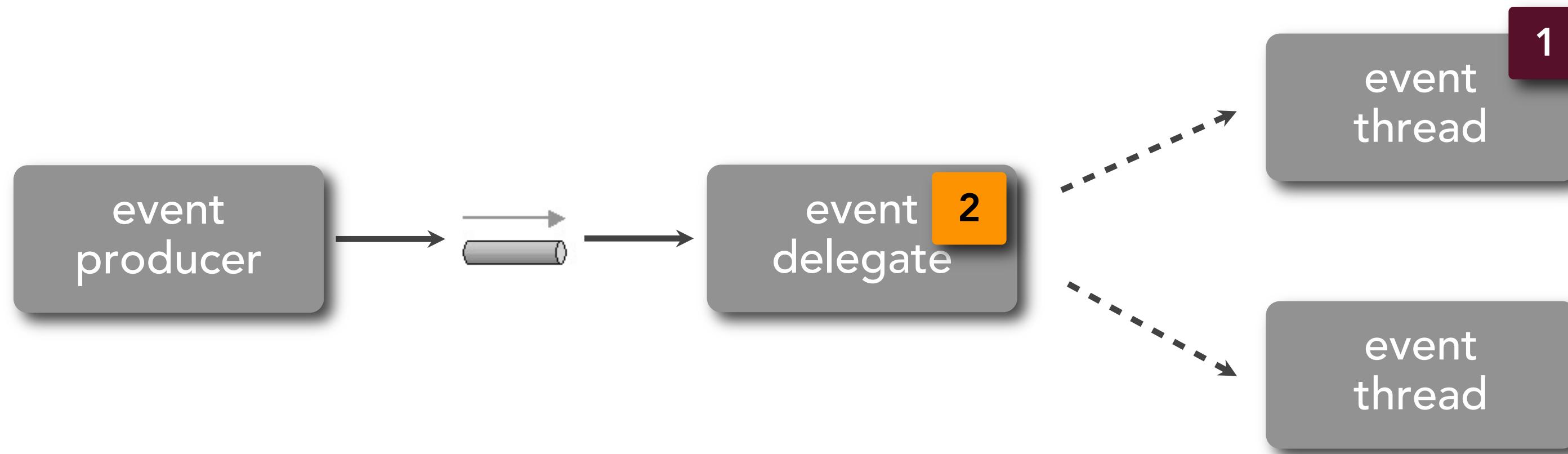
# thread delegate pattern



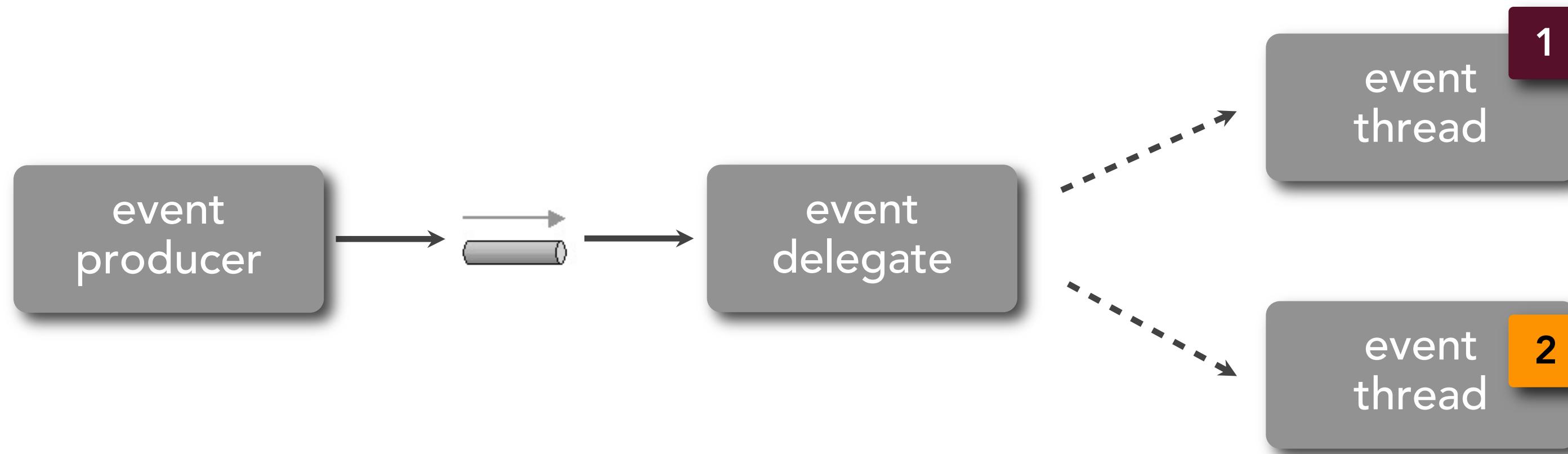
# thread delegate pattern



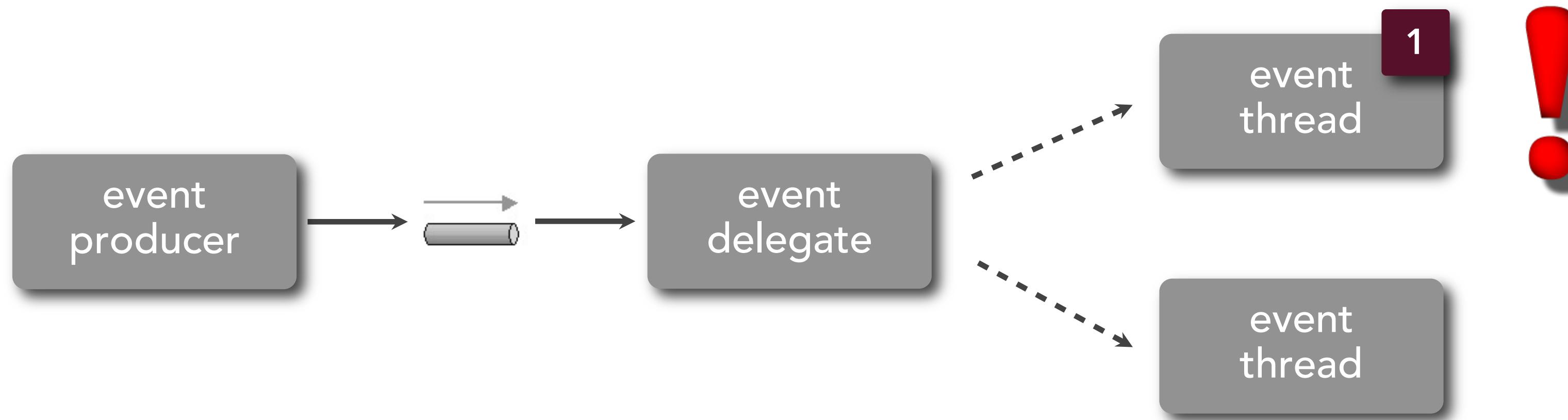
# thread delegate pattern



# thread delegate pattern

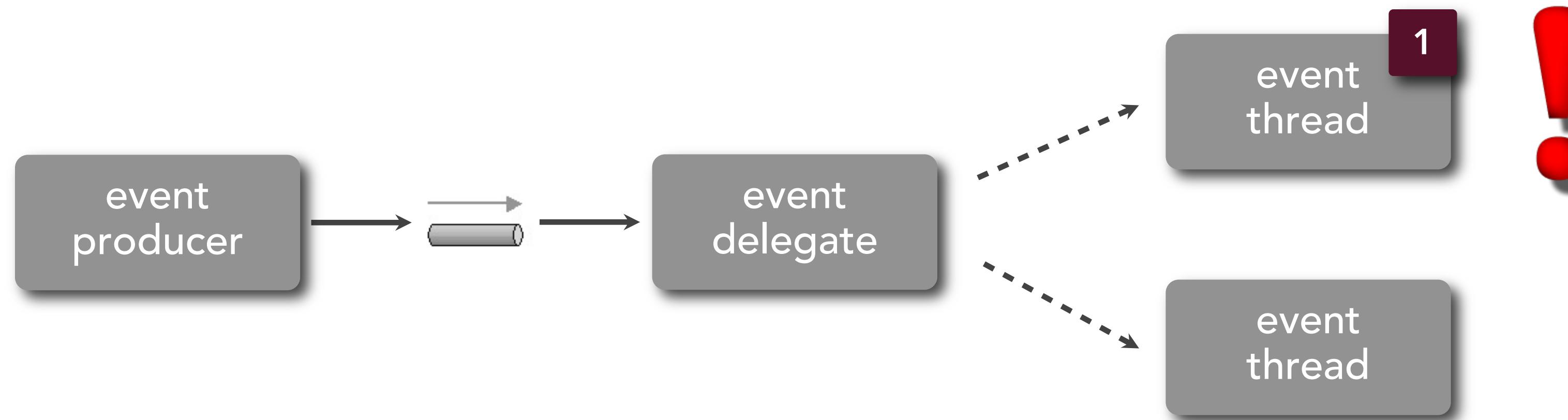


# thread delegate pattern



# thread delegate pattern

wait - isn't this IMPOSSIBLE?

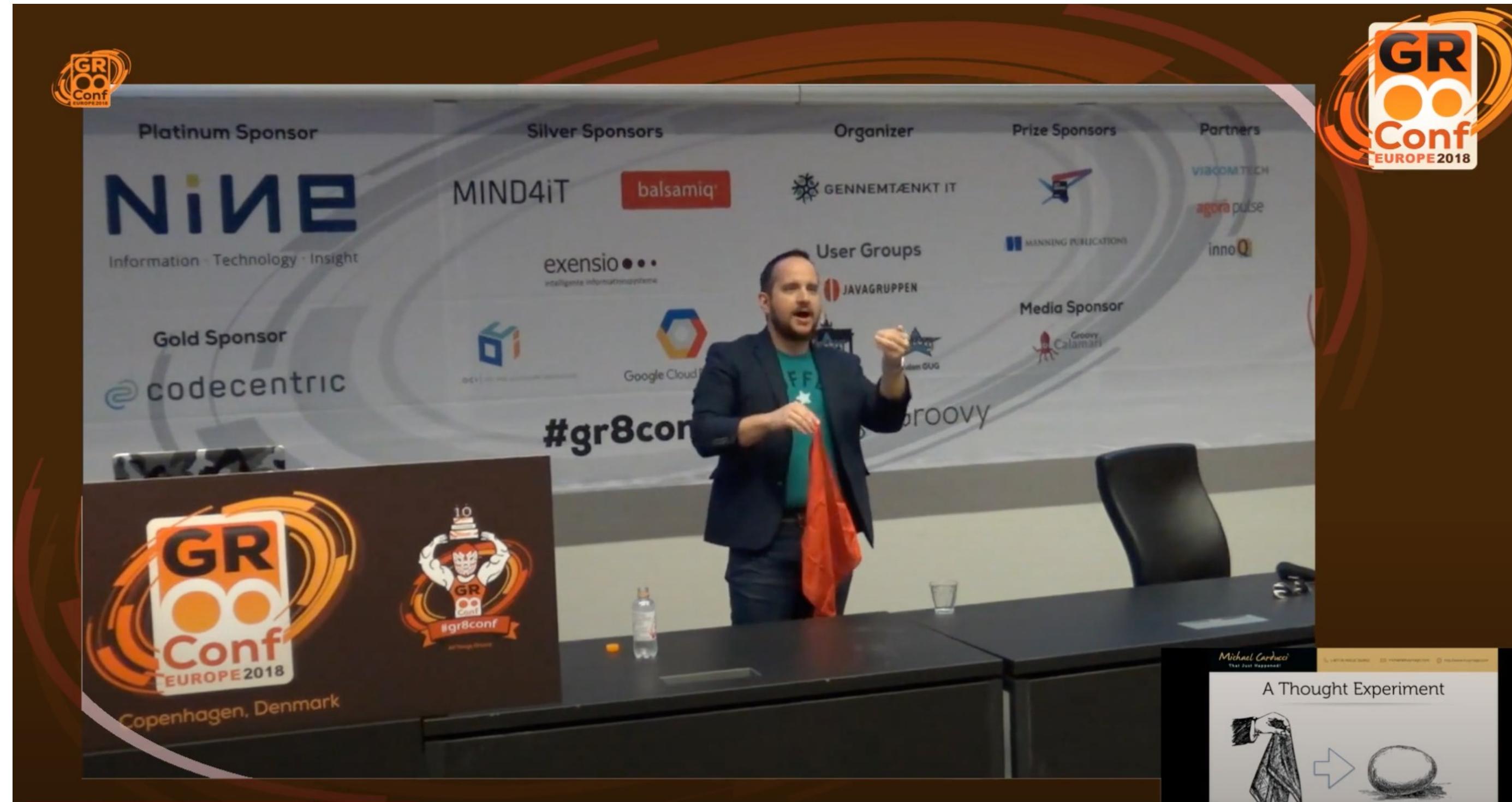


# thread delegate pattern

## The Art of the Impossible

<https://www.youtube.com/watch?v=Z4LRexmnik>

(15:10 - 21:40)



# thread delegate pattern

not every message must be ordered, but rather messages  
**within a specific context** must be ordered

# thread delegate pattern

not every message must be ordered, but rather messages  
**within a specific context** must be ordered

1. PLACE SELL GOOG A-136 2,000,000.00
2. PLACE BUY AAPL A-136 1,200,000.00
3. CANCEL BUY AAPL A-136 1,200,000.00

# thread delegate pattern

not every message must be ordered, but rather messages  
**within a specific context** must be ordered

1. PLACE SELL GOOG A-136 2,000,000.00
2. PLACE BUY AAPL A-136 1,200,000.00
3. CANCEL BUY AAPL A-136 1,200,000.00



# thread delegate pattern

not every message must be ordered, but rather messages  
**within a specific context** must be ordered

1. PLACE SELL GOOG A-136 2,000,000.00
2. PLACE BUY AAPL A-136 1,200,000.00
3. CANCEL BUY AAPL A-136 1,200,000.00



1. PLACE SELL AAPL A-136 2,000,000.00
2. PLACE SELL GOOG V-976 650,000.00
3. PLACE BUY ATT V-976 900,000.00
4. PLACE BUY IBM A-136 1,300,000.00
5. CANCEL BUY ATT V-976 900,000.00

# thread delegate pattern

not every message must be ordered, but rather messages  
**within a specific context** must be ordered

1. PLACE SELL GOOG A-136 2,000,000.00
2. PLACE BUY AAPL A-136 1,200,000.00
3. CANCEL BUY AAPL A-136 1,200,000.00



1. PLACE SELL AAPL A-136 2,000,000.00
2. PLACE SELL GOOG V-976 650,000.00
3. PLACE BUY ATT V-976 900,000.00
4. PLACE BUY IBM A-136 1,300,000.00
5. CANCEL BUY ATT V-976 900,000.00



# thread delegate pattern

not every message must be ordered, but rather messages  
**within a specific context** must be ordered

1. PLACE SELL GOOG A-136 2,000,000.00
2. PLACE BUY AAPL A-136 1,200,000.00
3. CANCEL BUY AAPL A-136 1,200,000.00

→ 1, 2, 3

1. PLACE SELL AAPL A-136 2,000,000.00
2. PLACE SELL GOOG V-976 650,000.00
3. PLACE BUY ATT V-976 900,000.00
4. PLACE BUY IBM A-136 1,300,000.00
5. CANCEL BUY ATT V-976 900,000.00

→ 2, 3, 5

# thread delegate pattern

not every message must be ordered, but rather messages  
**within a specific context** must be ordered

1. PLACE SELL GOOG A-136 2,000,000.00
2. PLACE BUY AAPL A-136 1,200,000.00
3. CANCEL BUY AAPL A-136 1,200,000.00

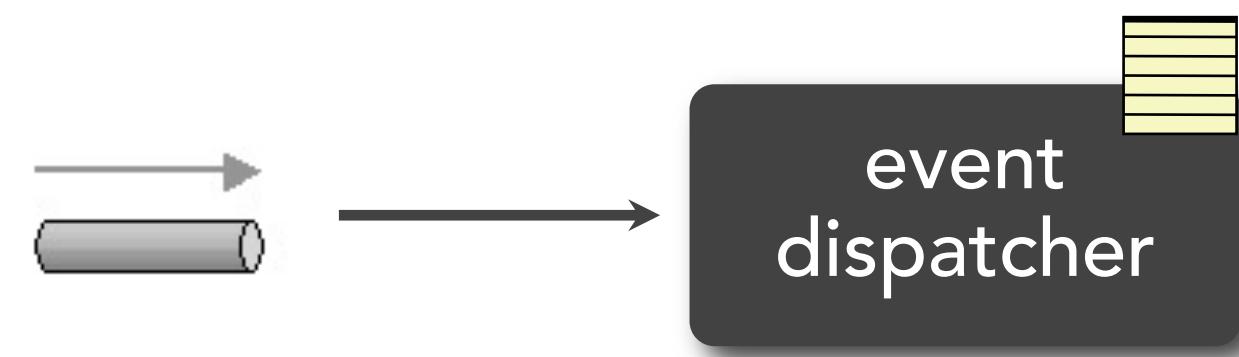
→ 1, 2, 3

1. PLACE SELL AAPL A-136 2,000,000.00
2. PLACE SELL GOOG V-976 650,000.00
3. PLACE BUY ATT V-976 900,000.00
4. PLACE BUY IBM A-136 1,300,000.00
5. CANCEL BUY ATT V-976 900,000.00

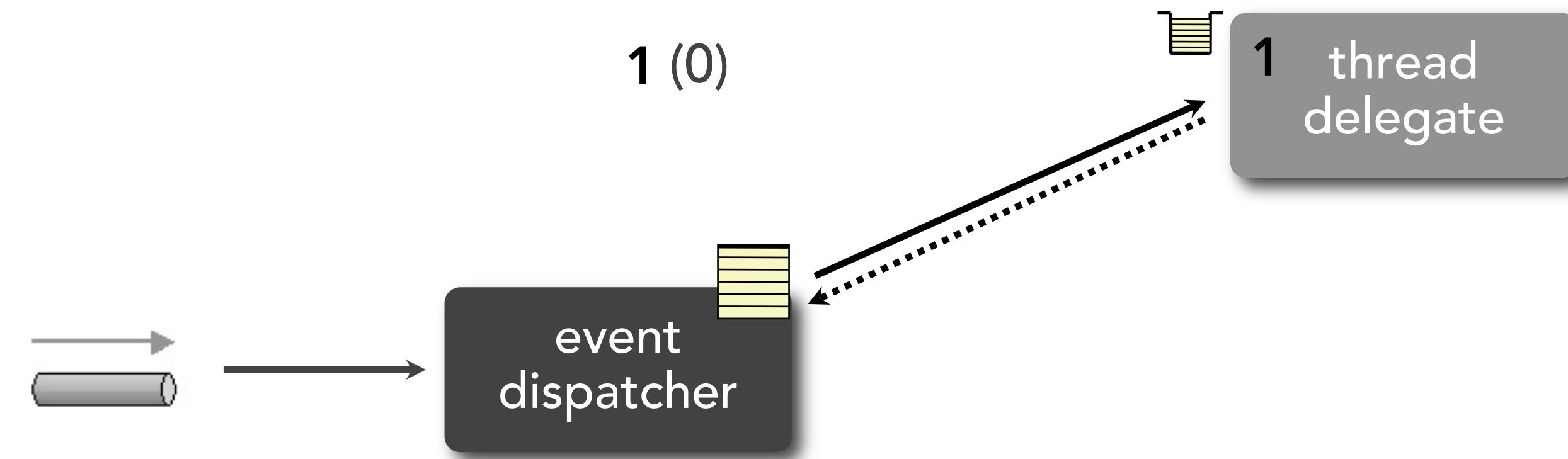
→ 1, 4

→ 2, 3, 5

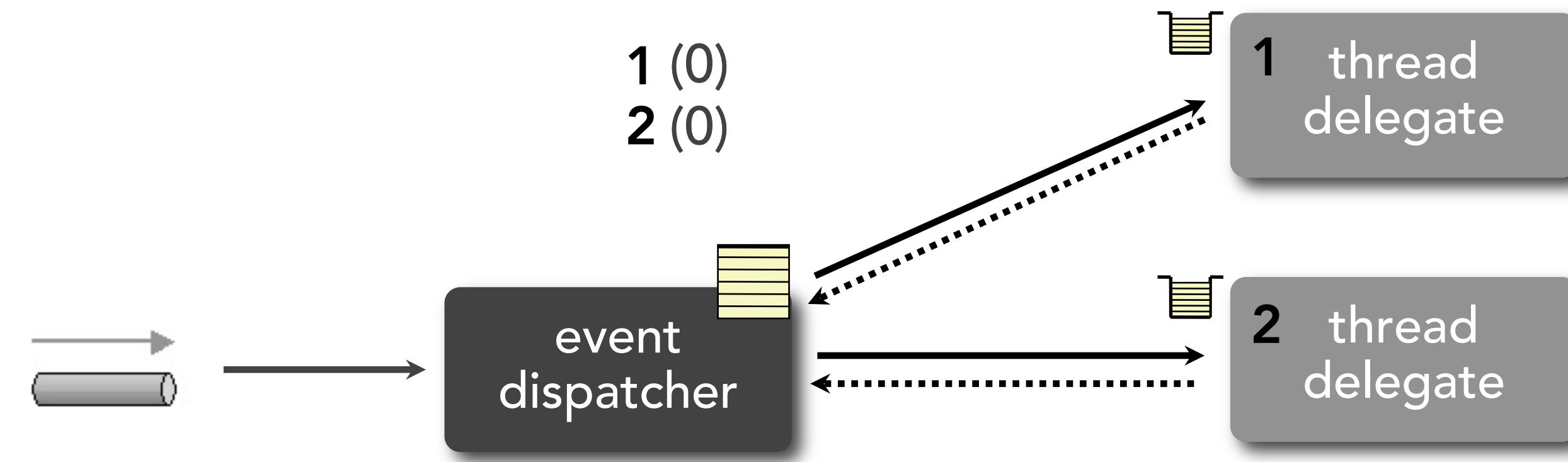
# thread delegate pattern



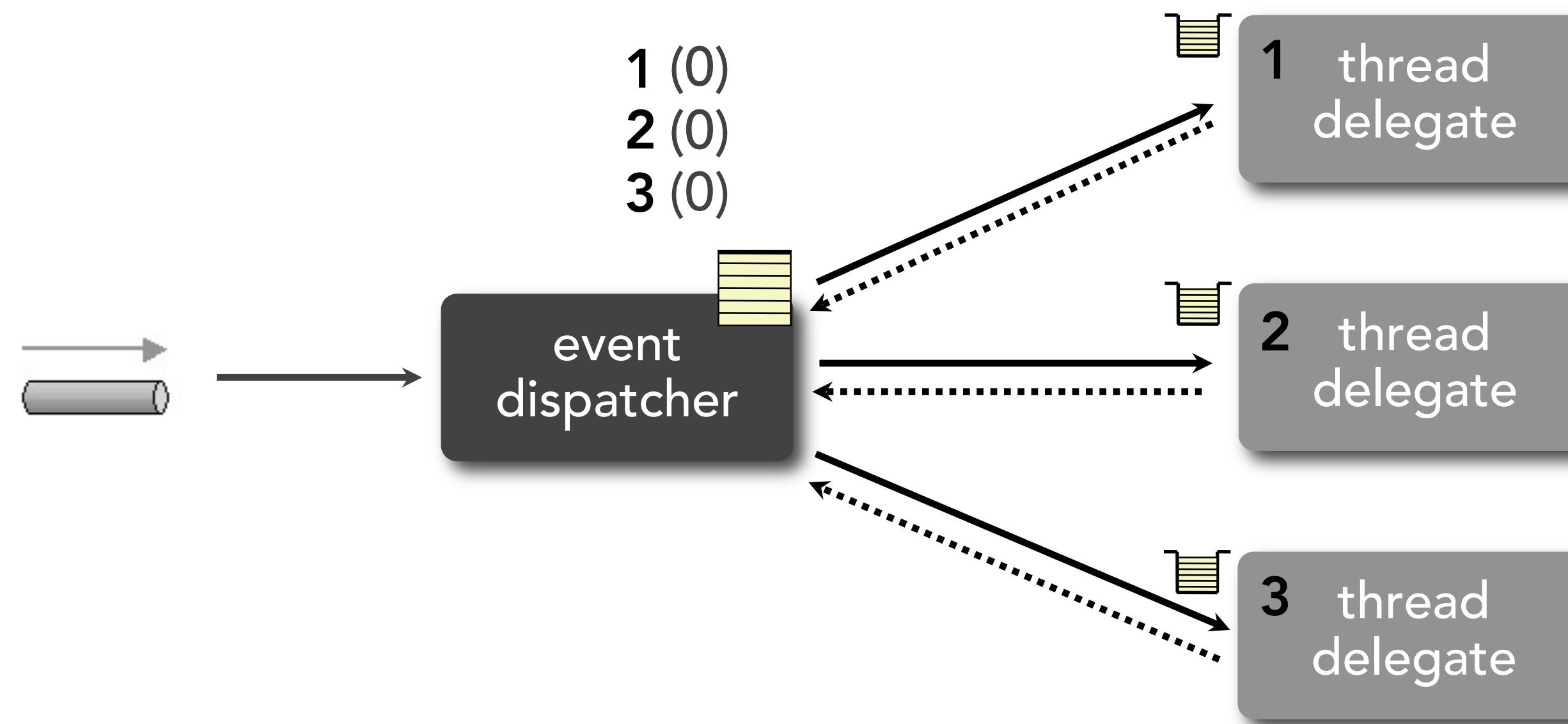
# thread delegate pattern



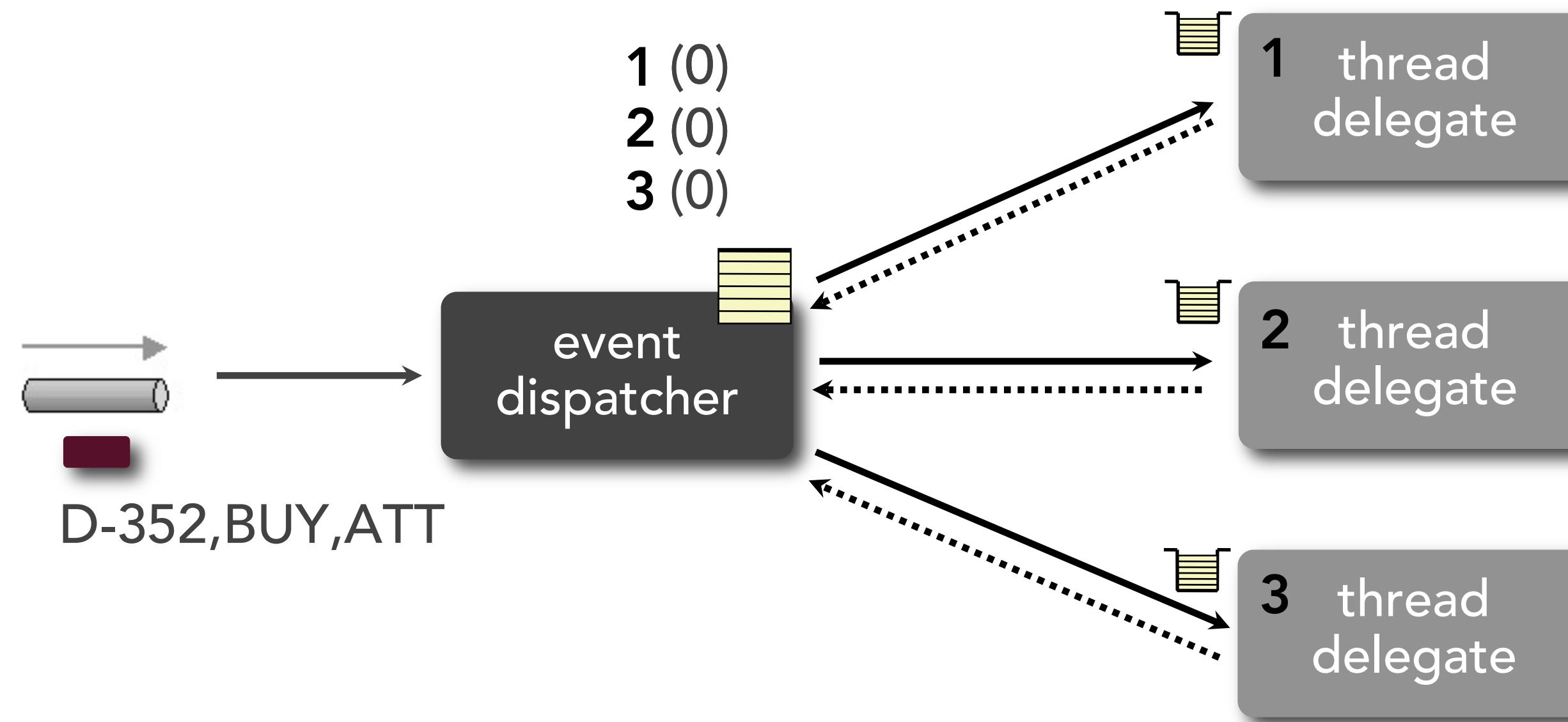
# thread delegate pattern



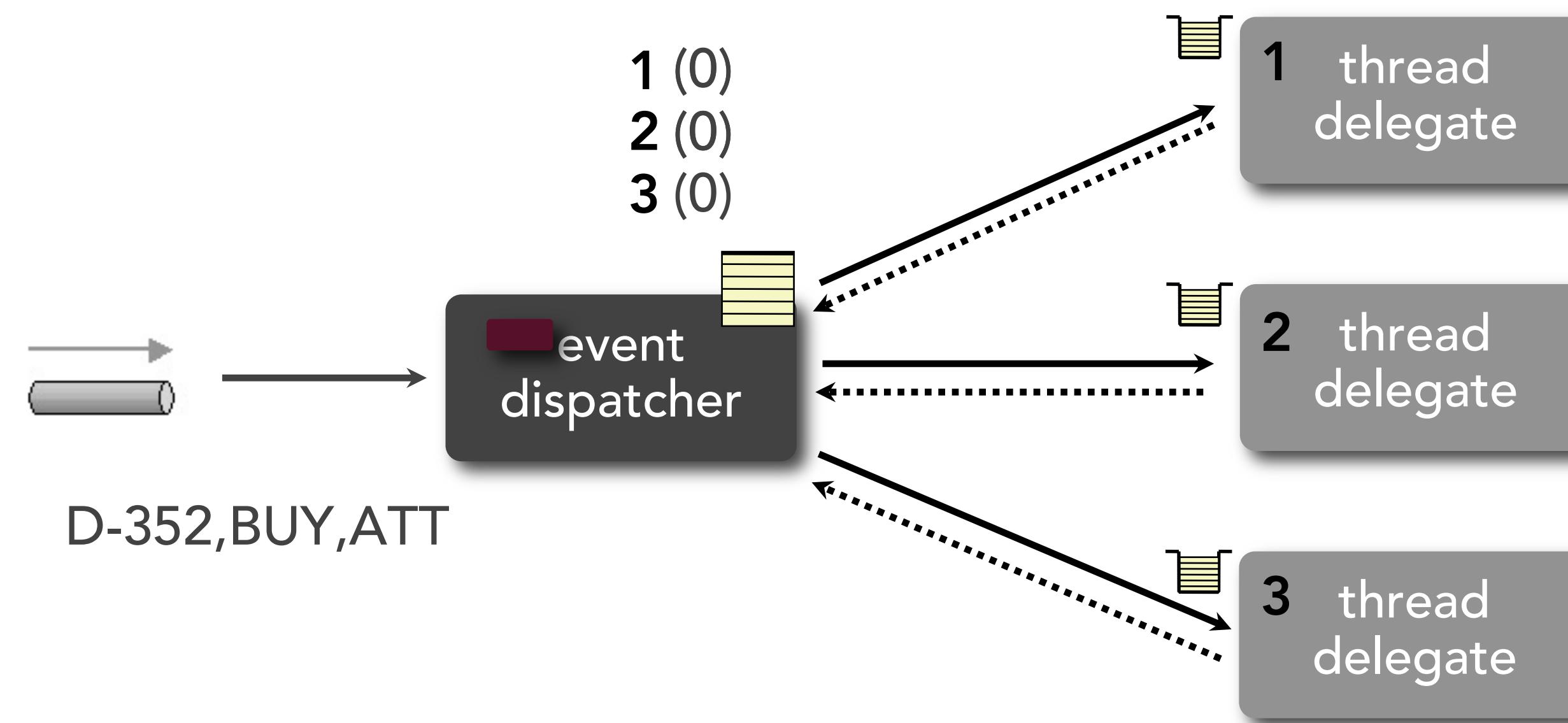
# thread delegate pattern



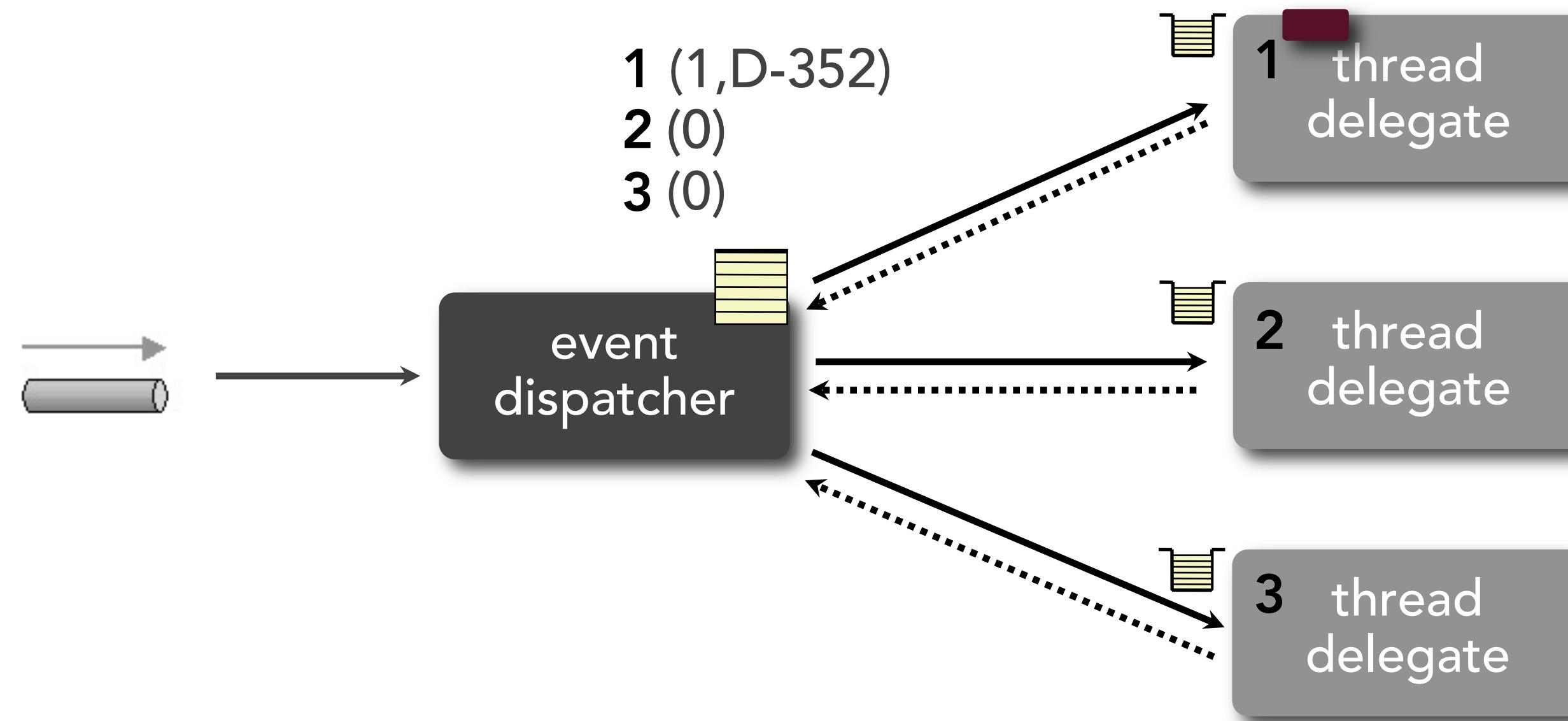
# thread delegate pattern



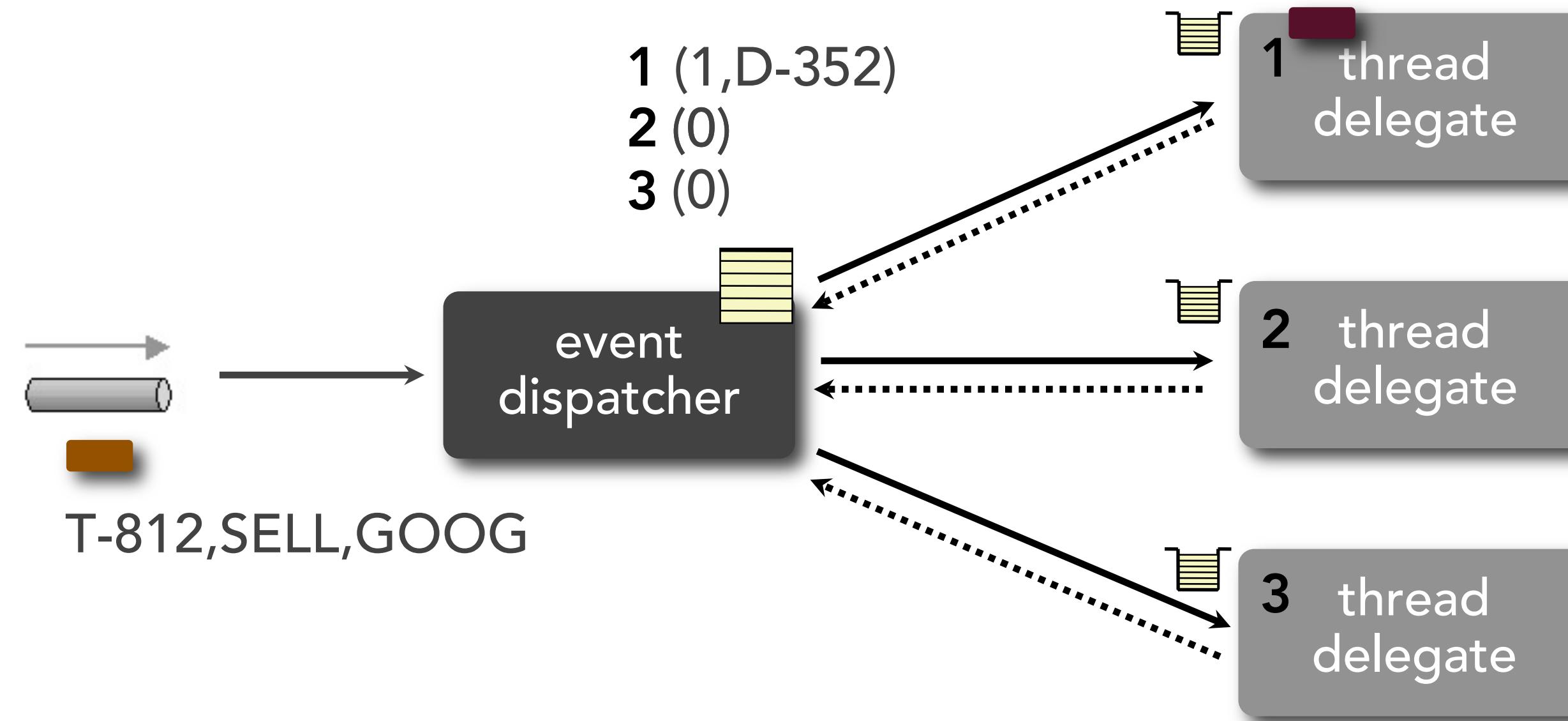
# thread delegate pattern



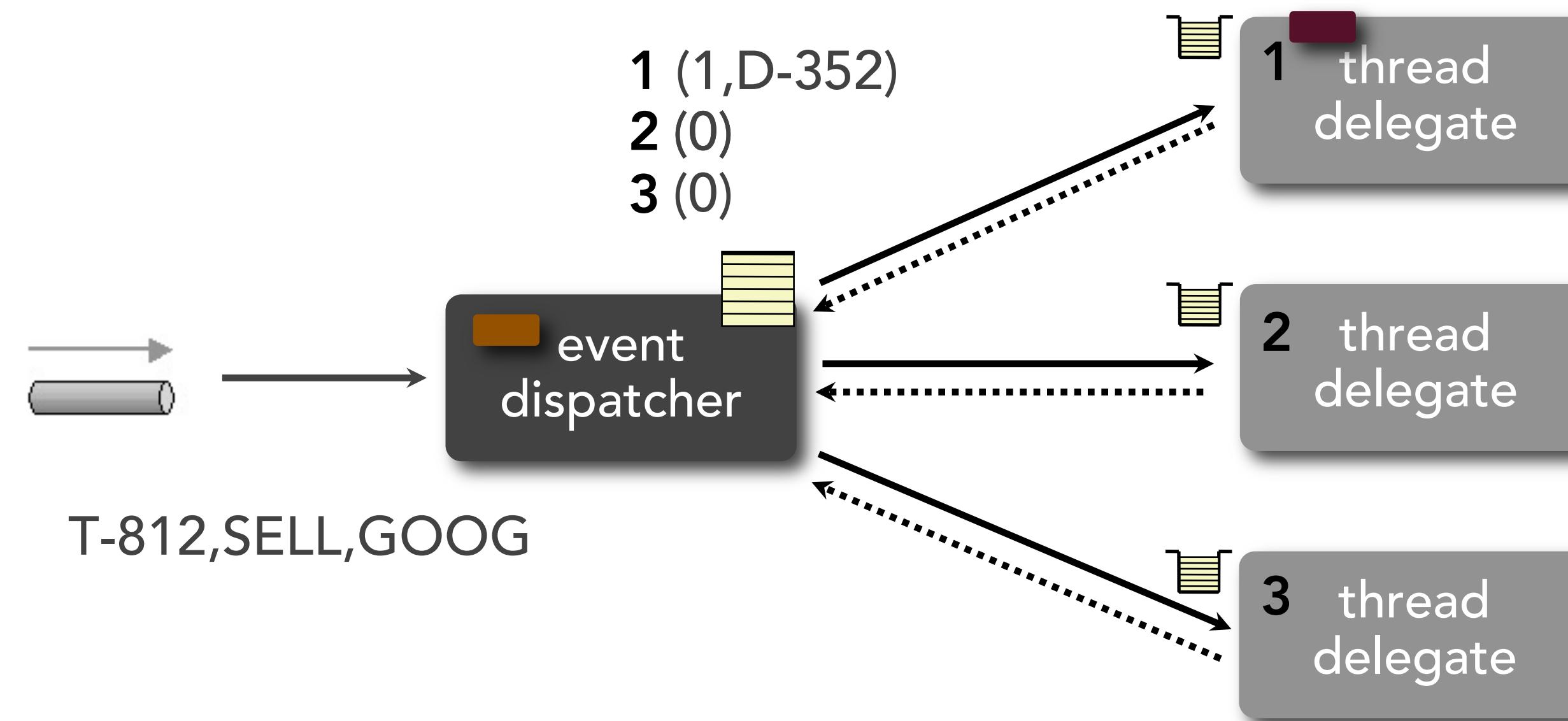
# thread delegate pattern



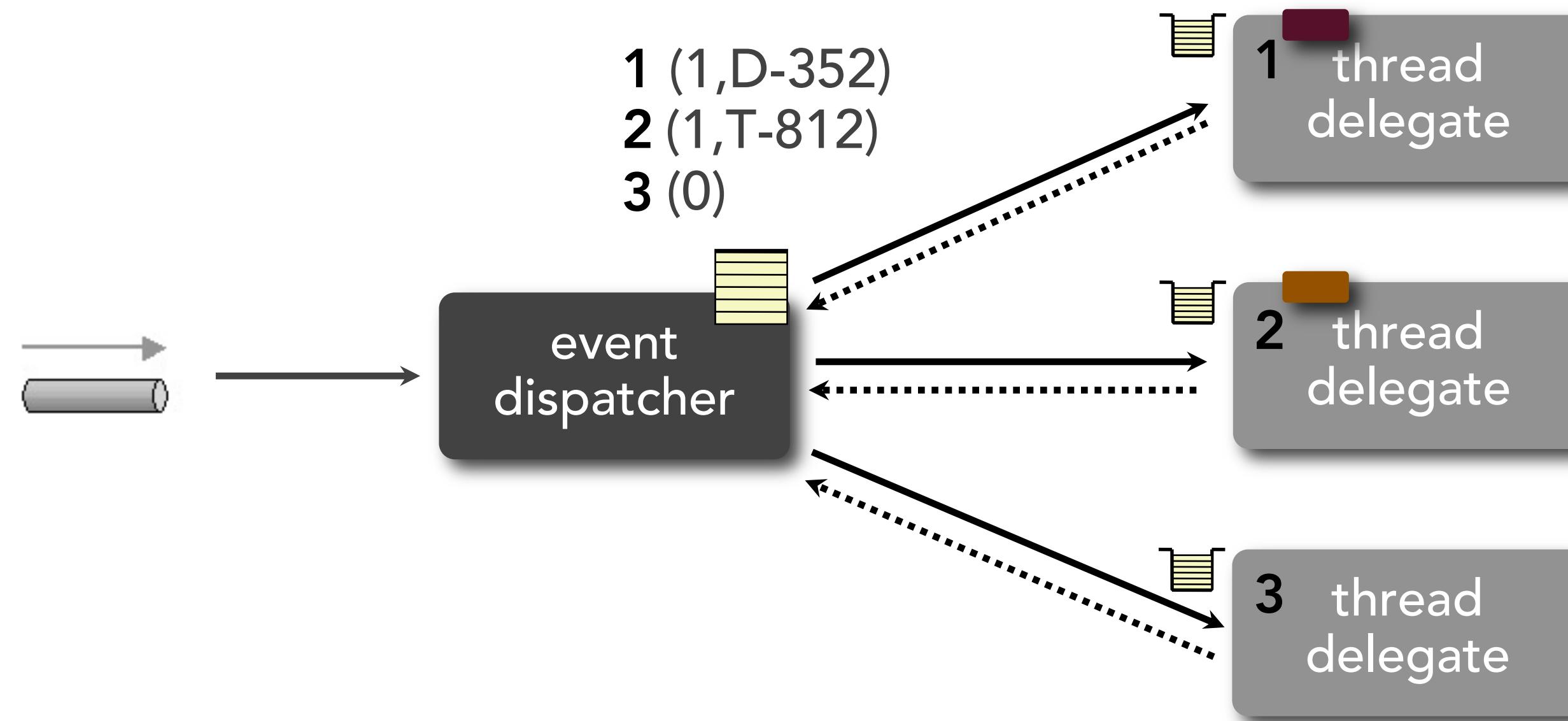
# thread delegate pattern



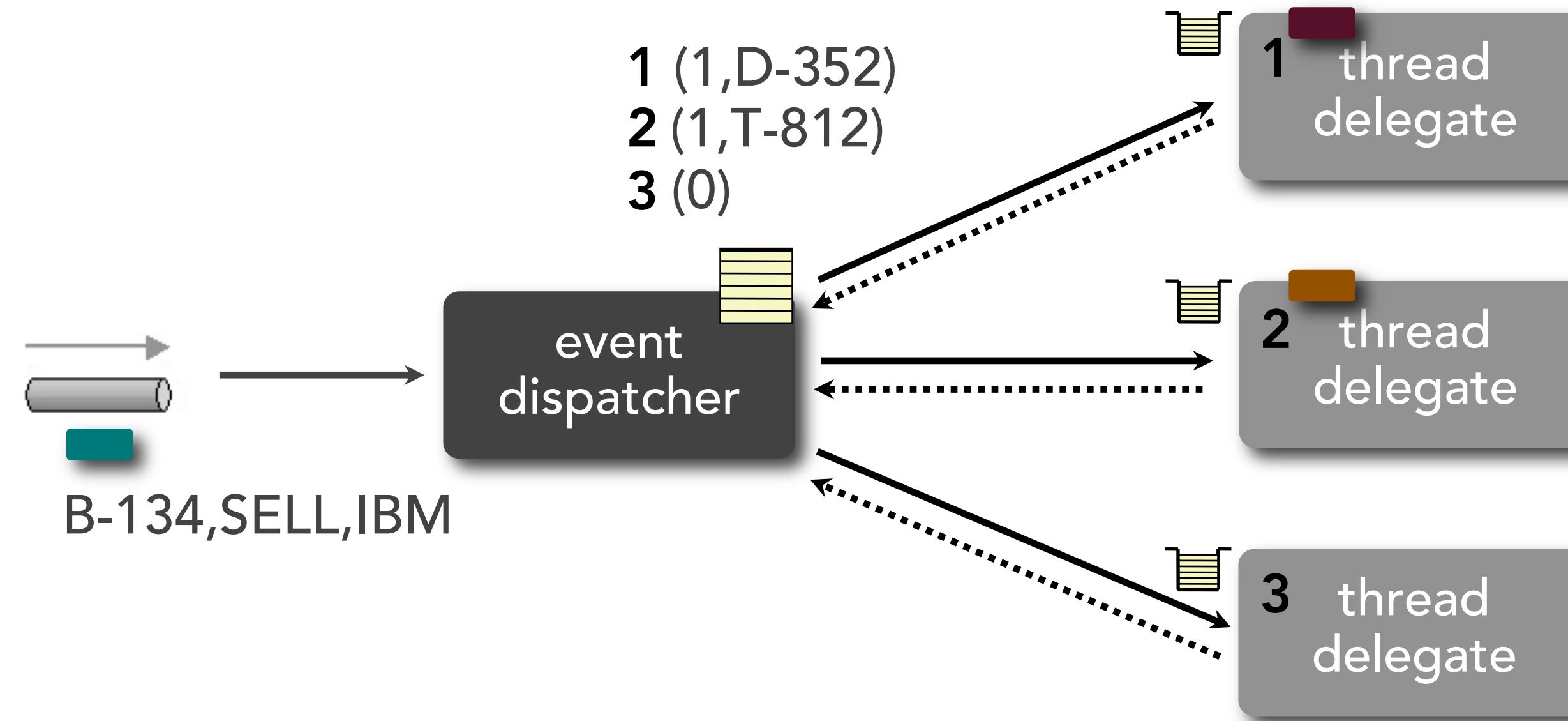
# thread delegate pattern



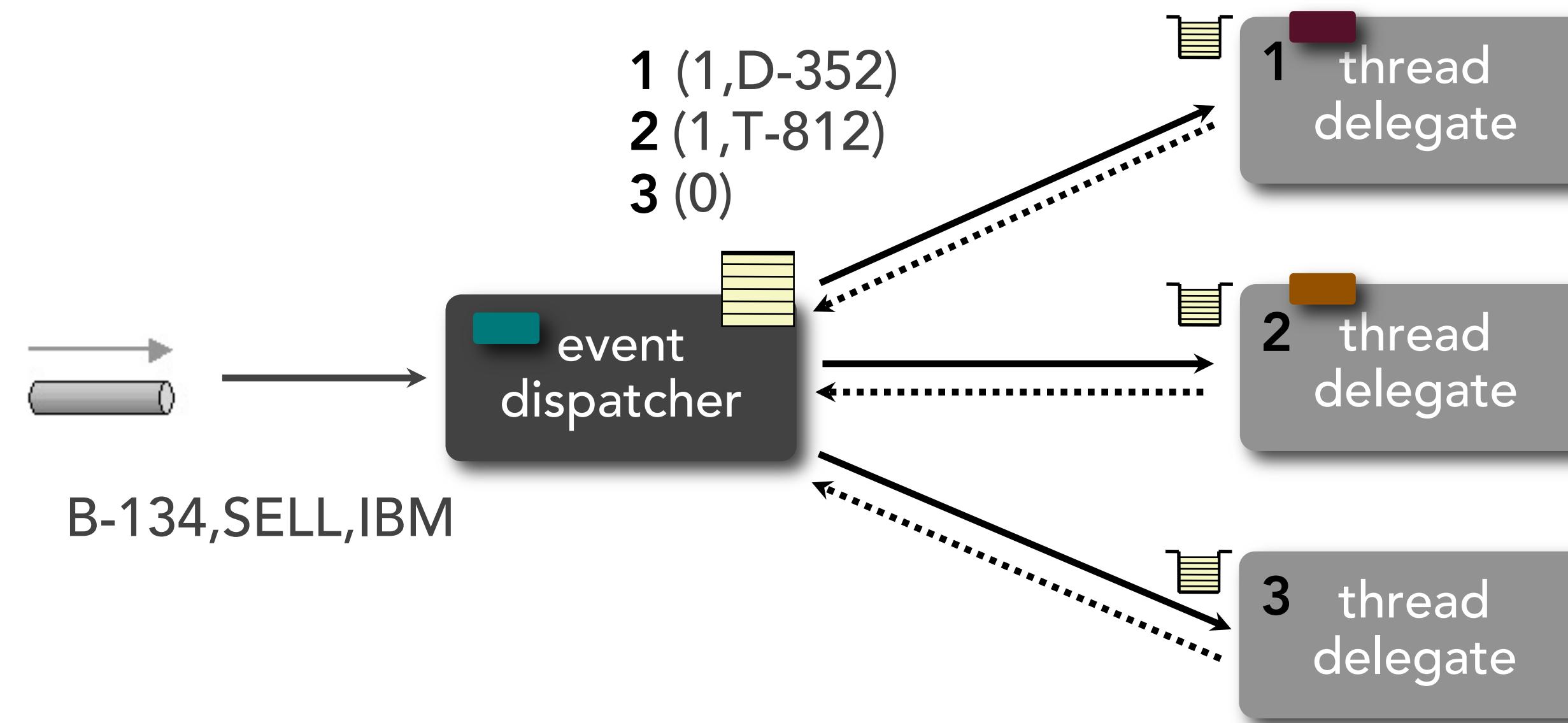
# thread delegate pattern



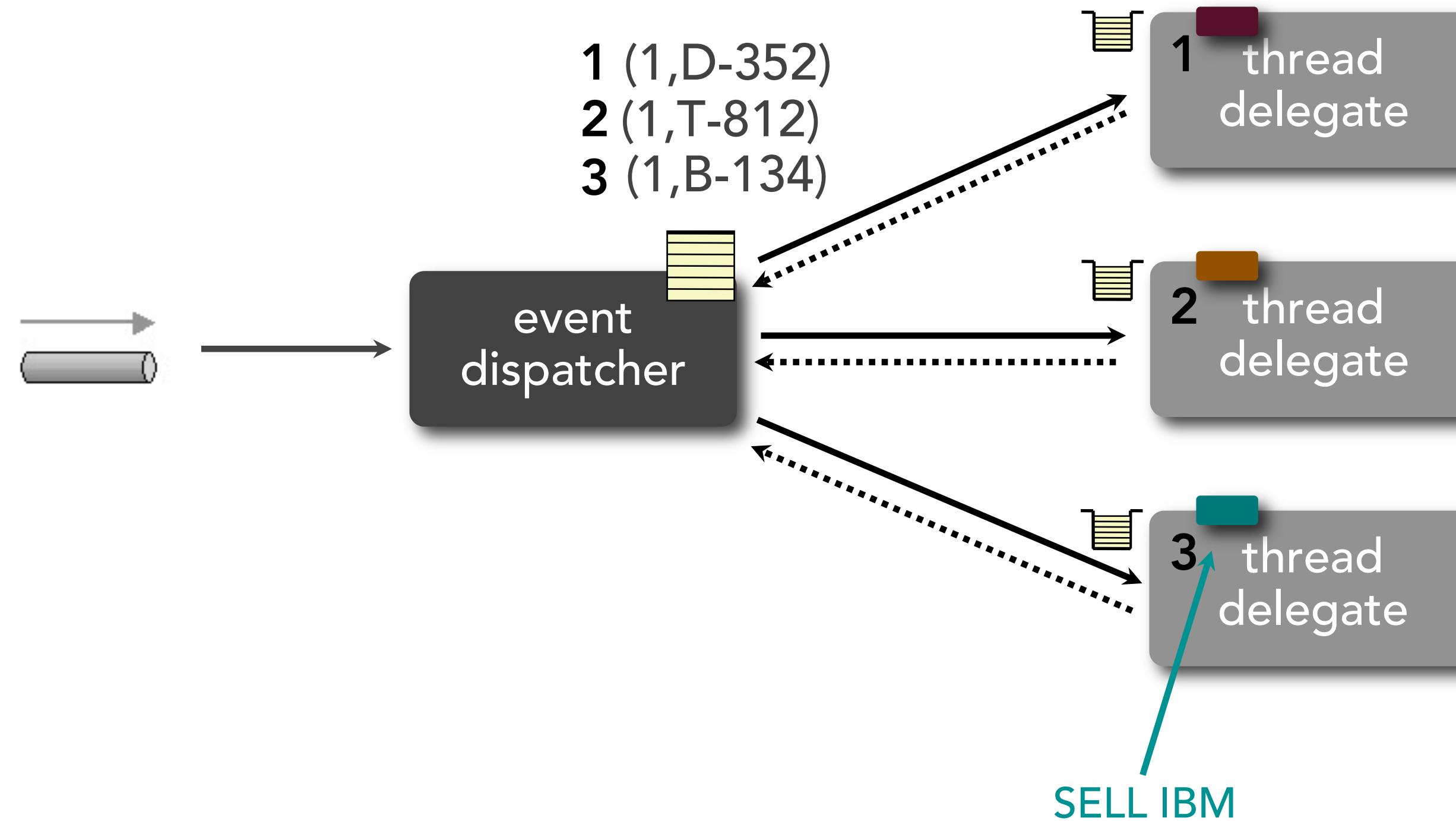
# thread delegate pattern



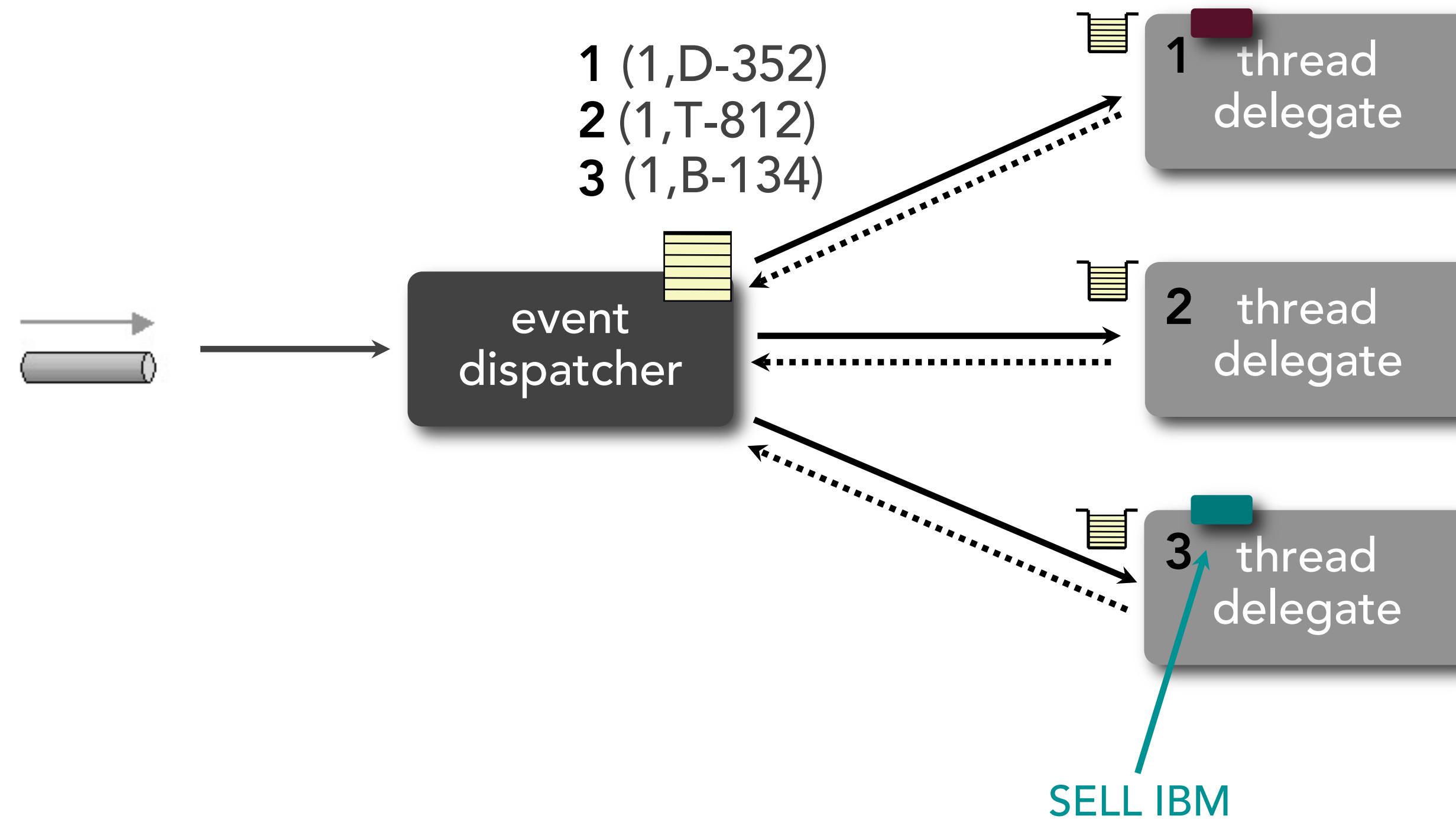
# thread delegate pattern



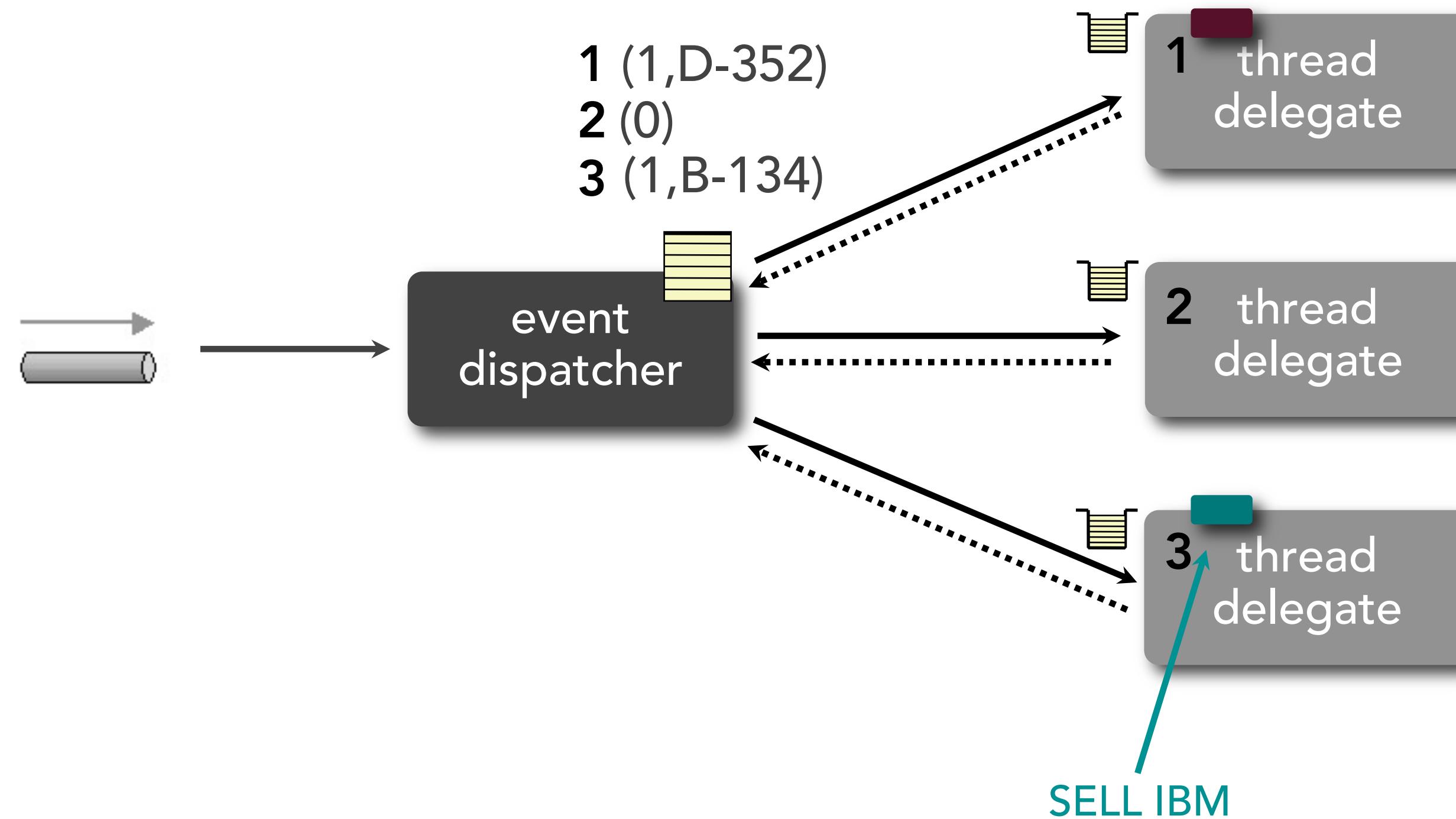
# thread delegate pattern



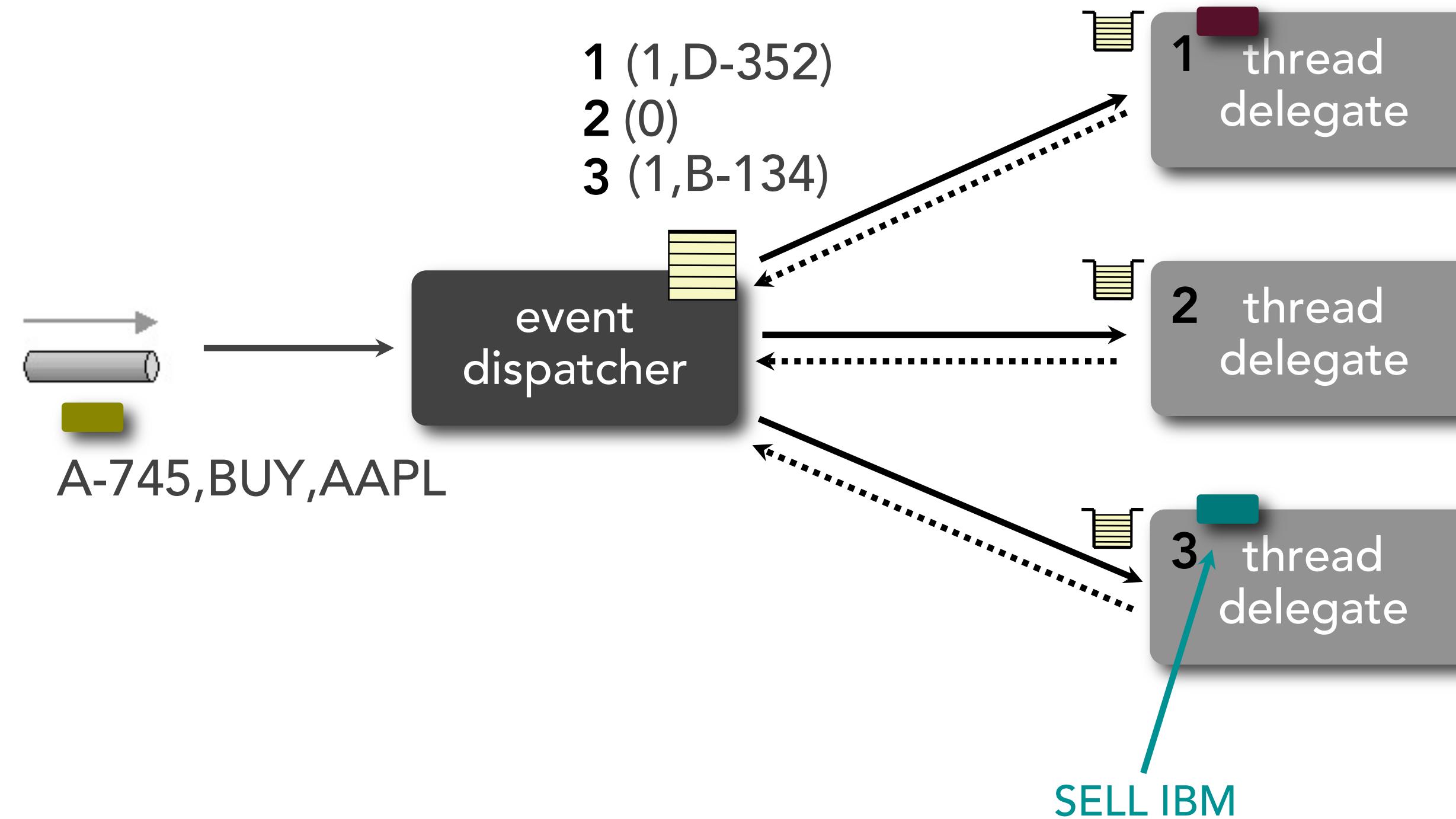
# thread delegate pattern



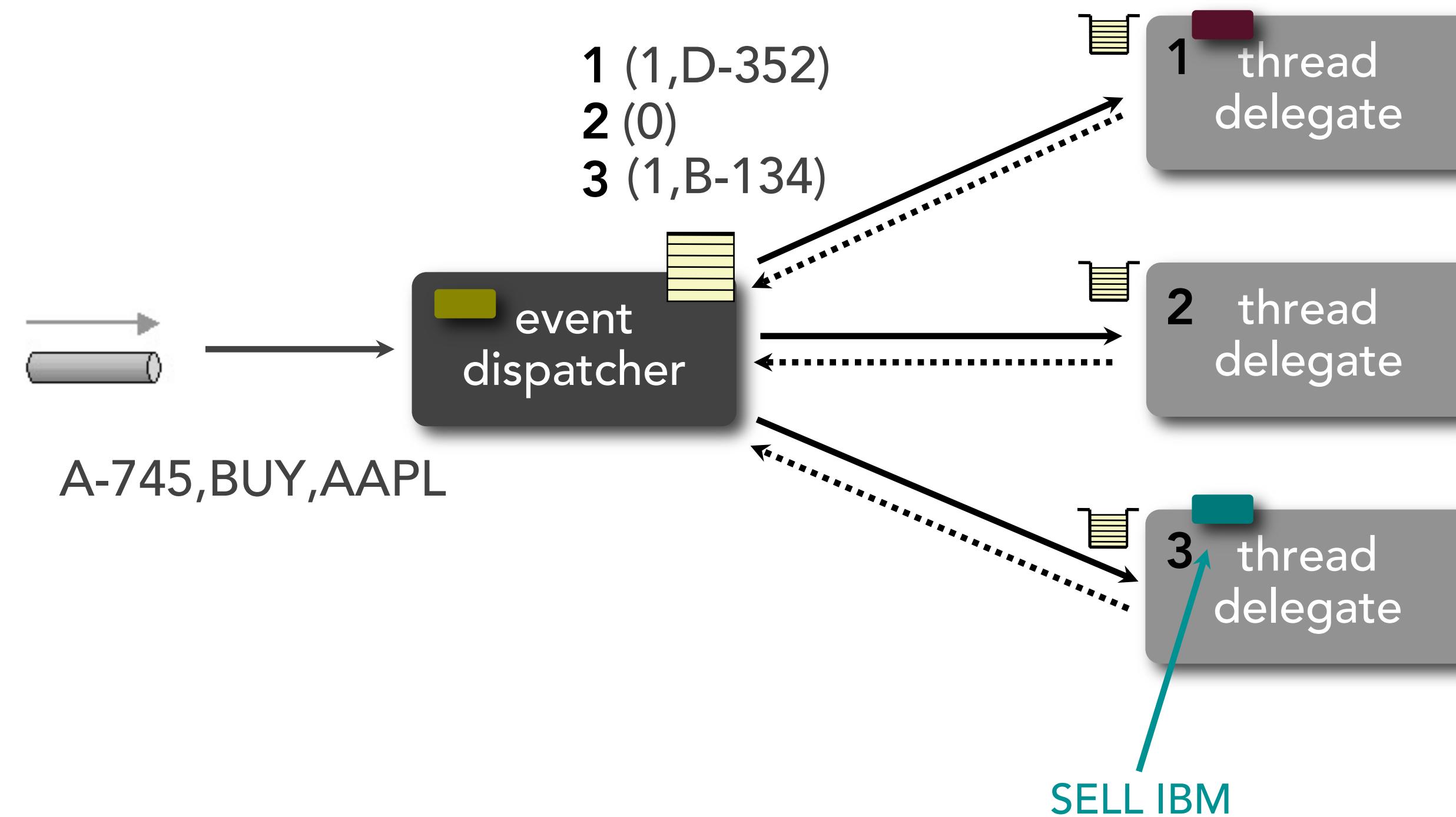
# thread delegate pattern



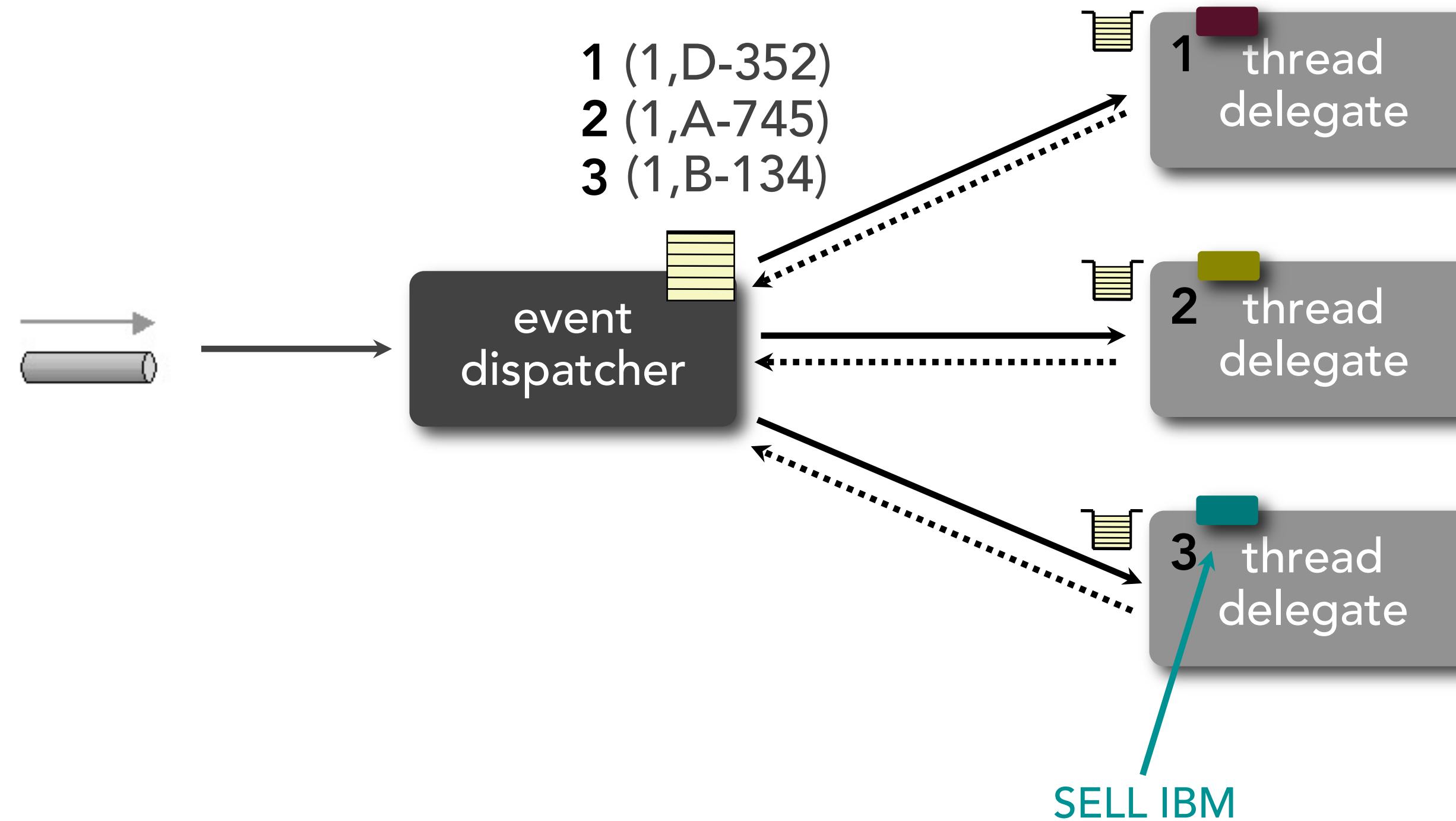
# thread delegate pattern



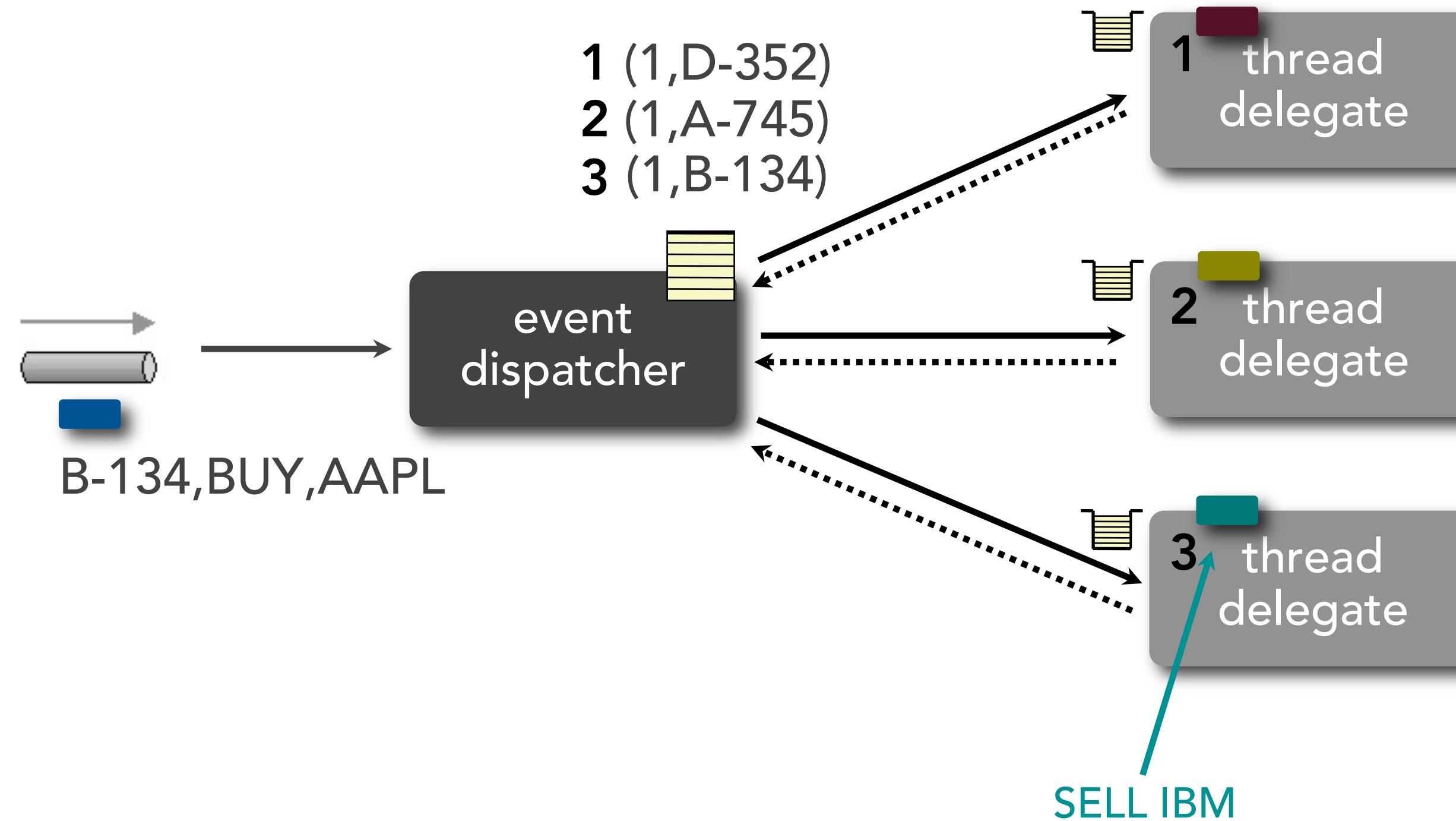
# thread delegate pattern



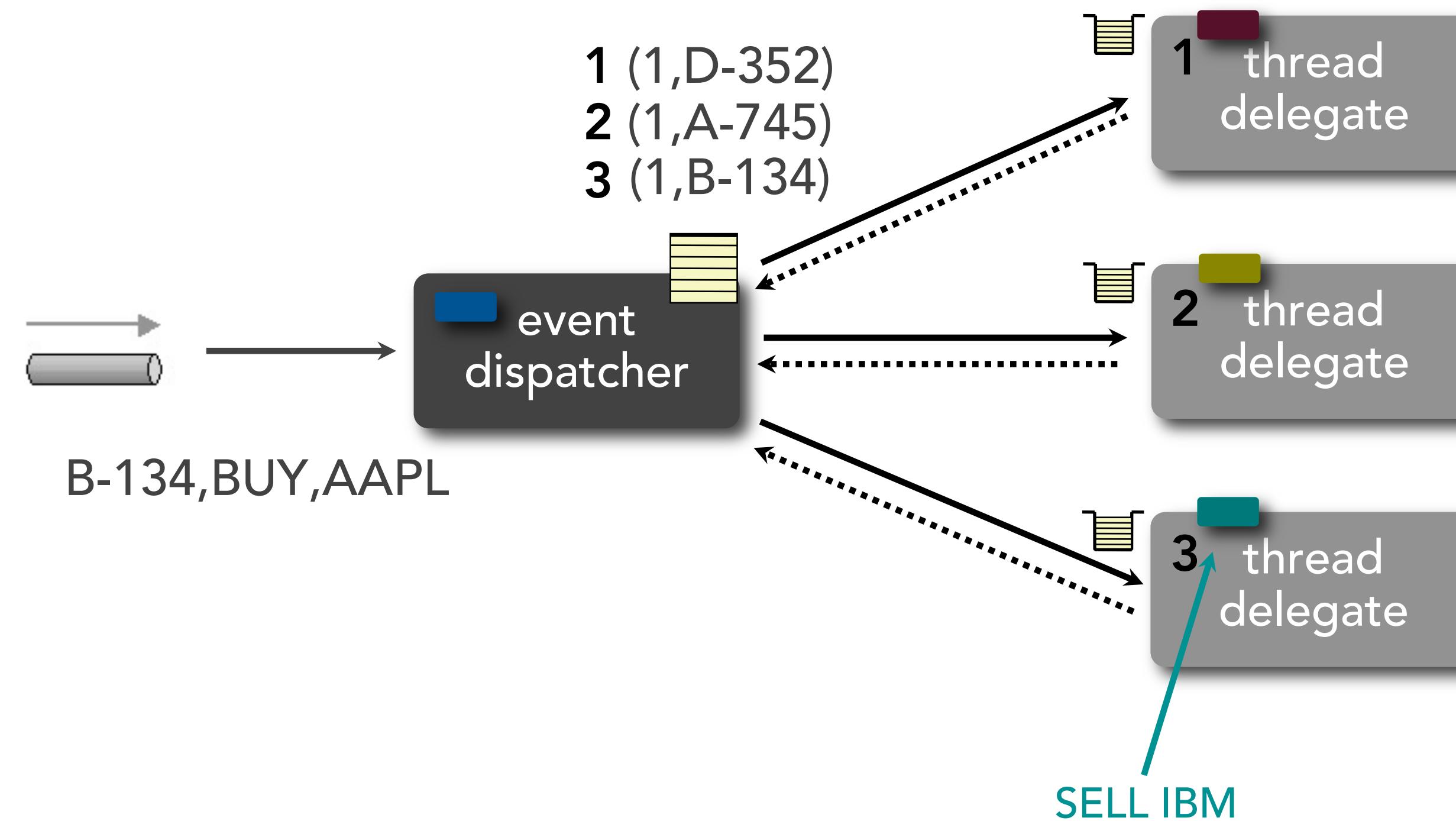
# thread delegate pattern



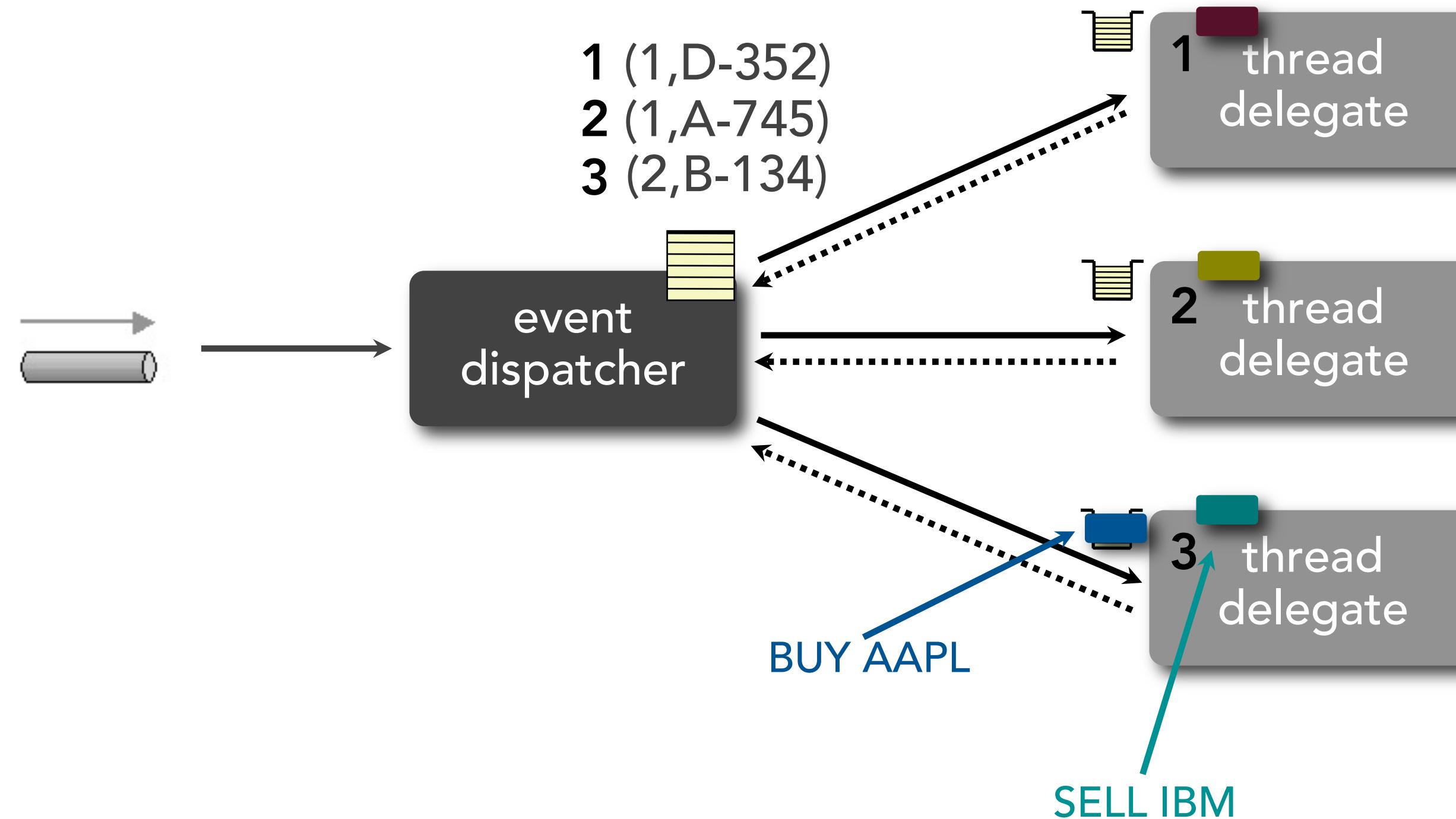
# thread delegate pattern



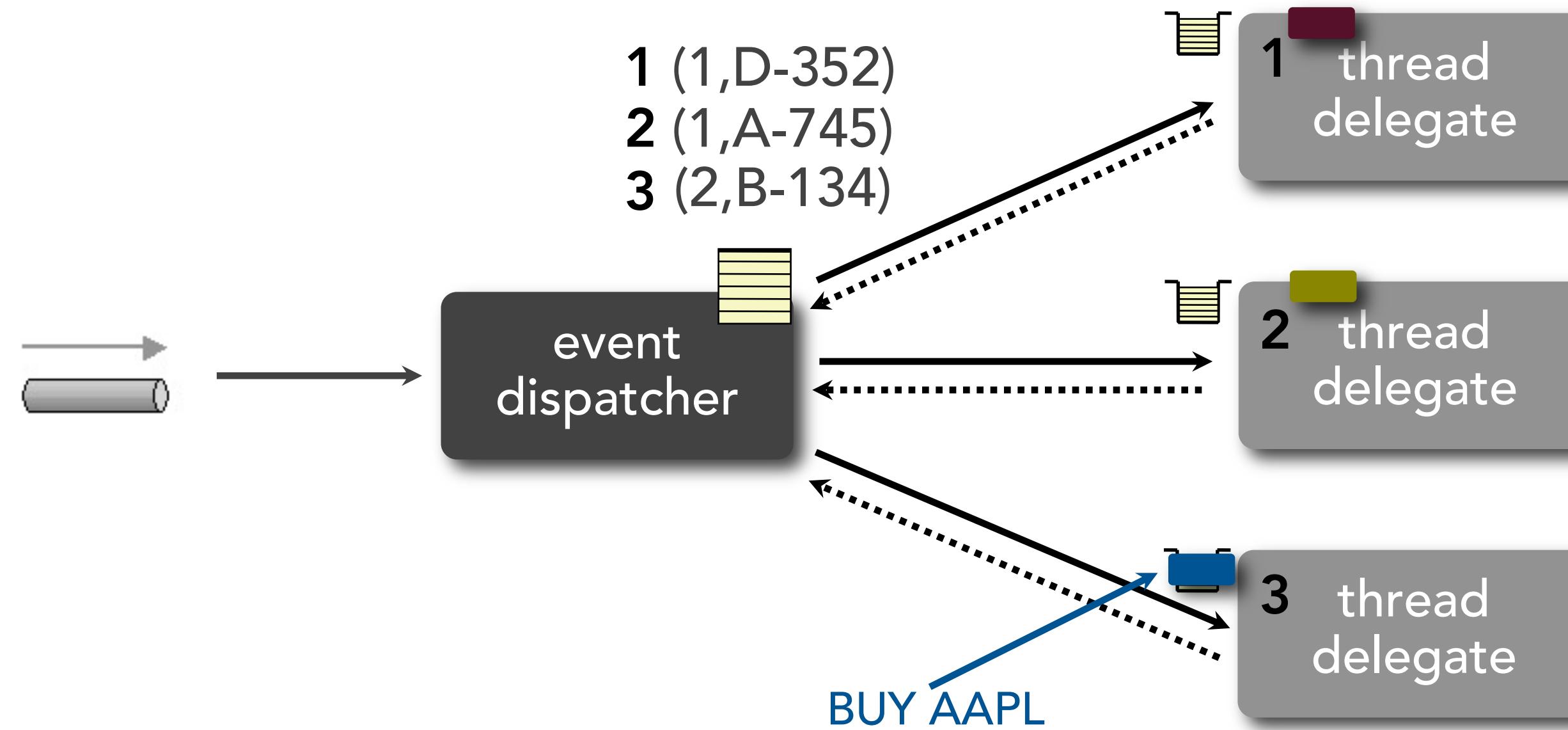
# thread delegate pattern



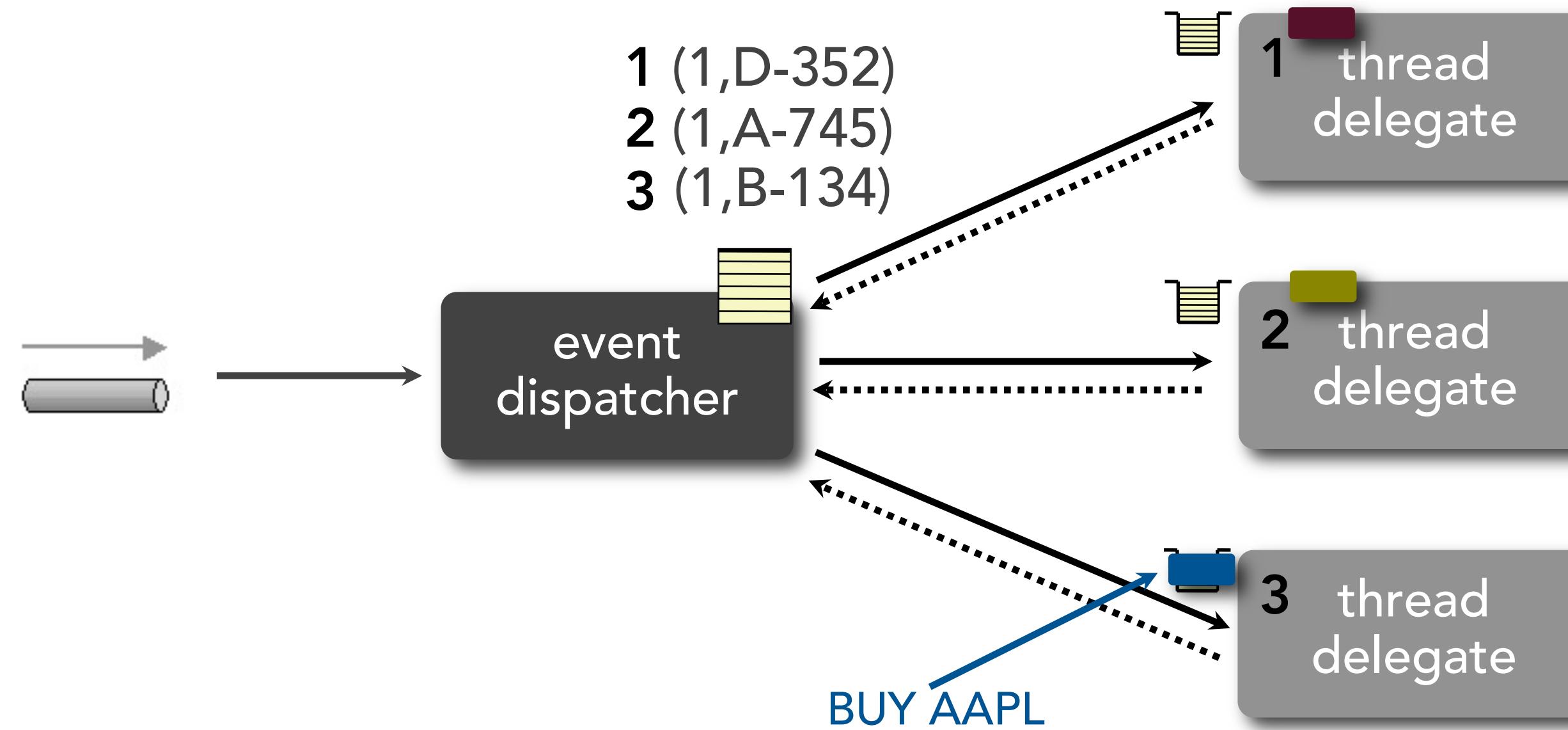
# thread delegate pattern



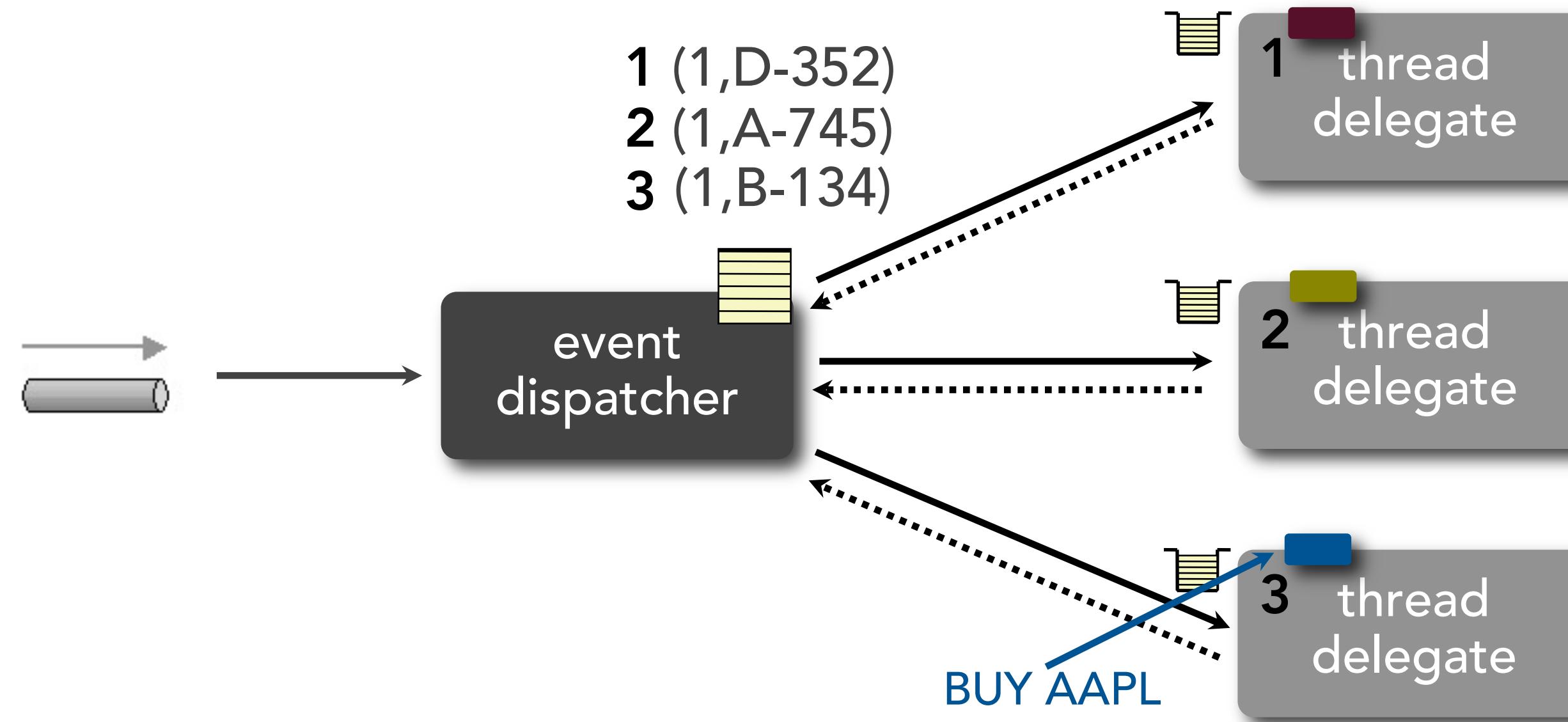
# thread delegate pattern



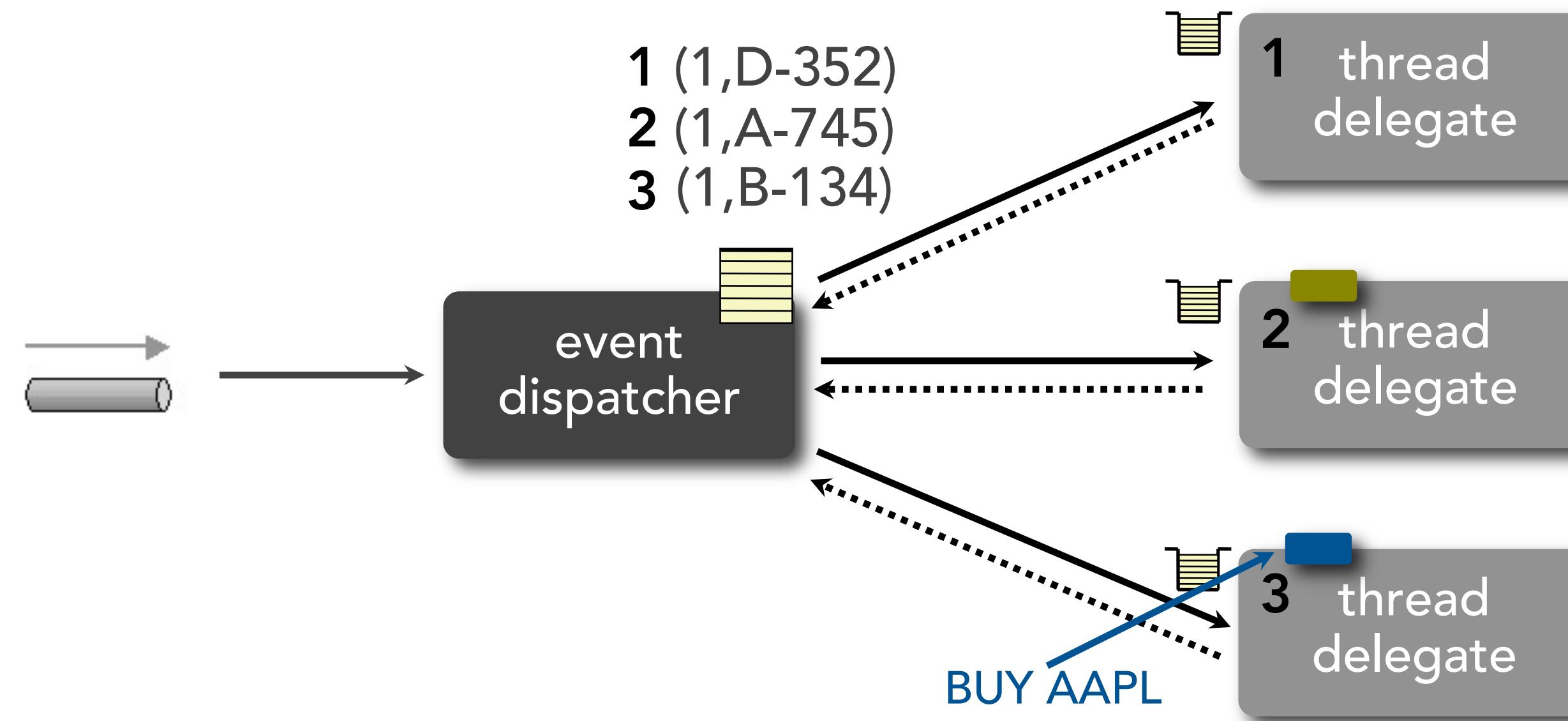
# thread delegate pattern



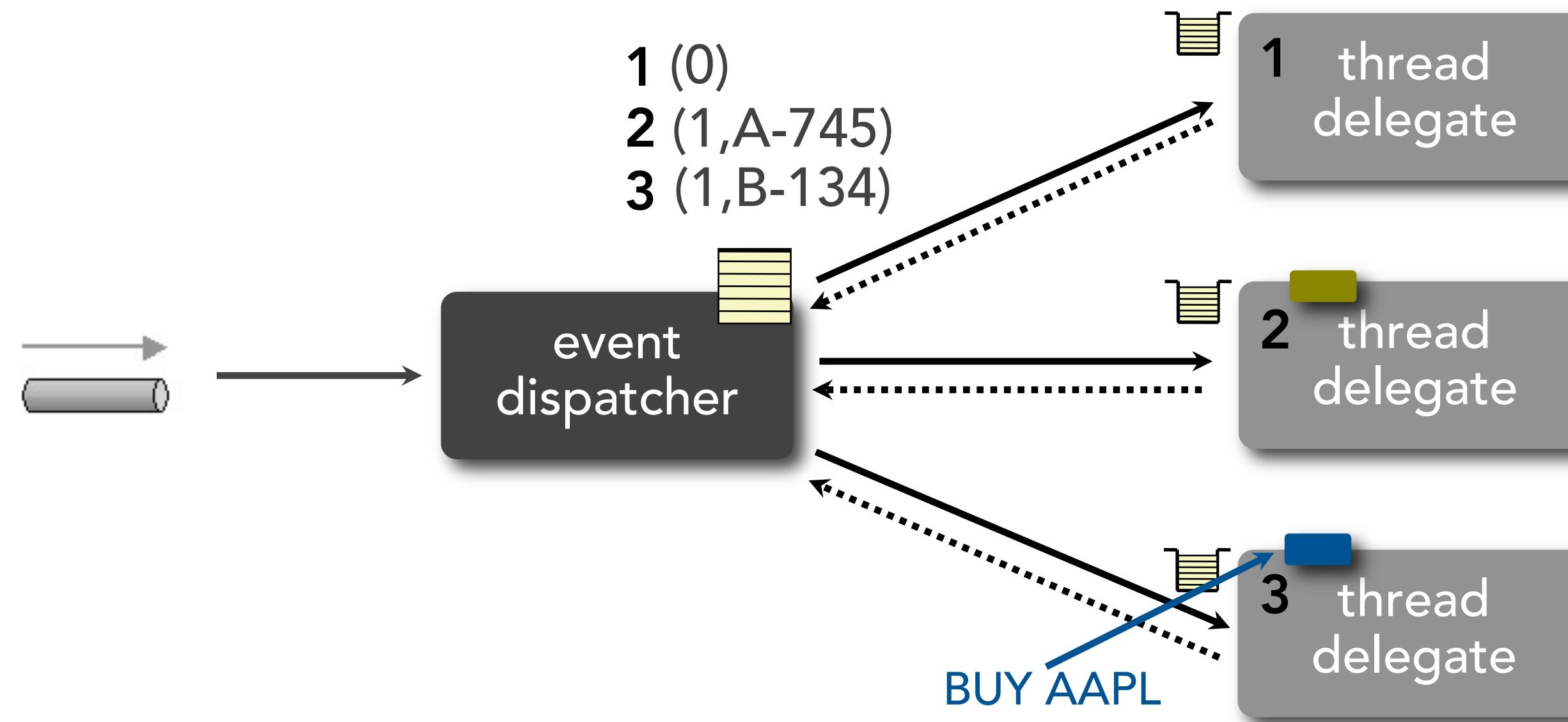
# thread delegate pattern



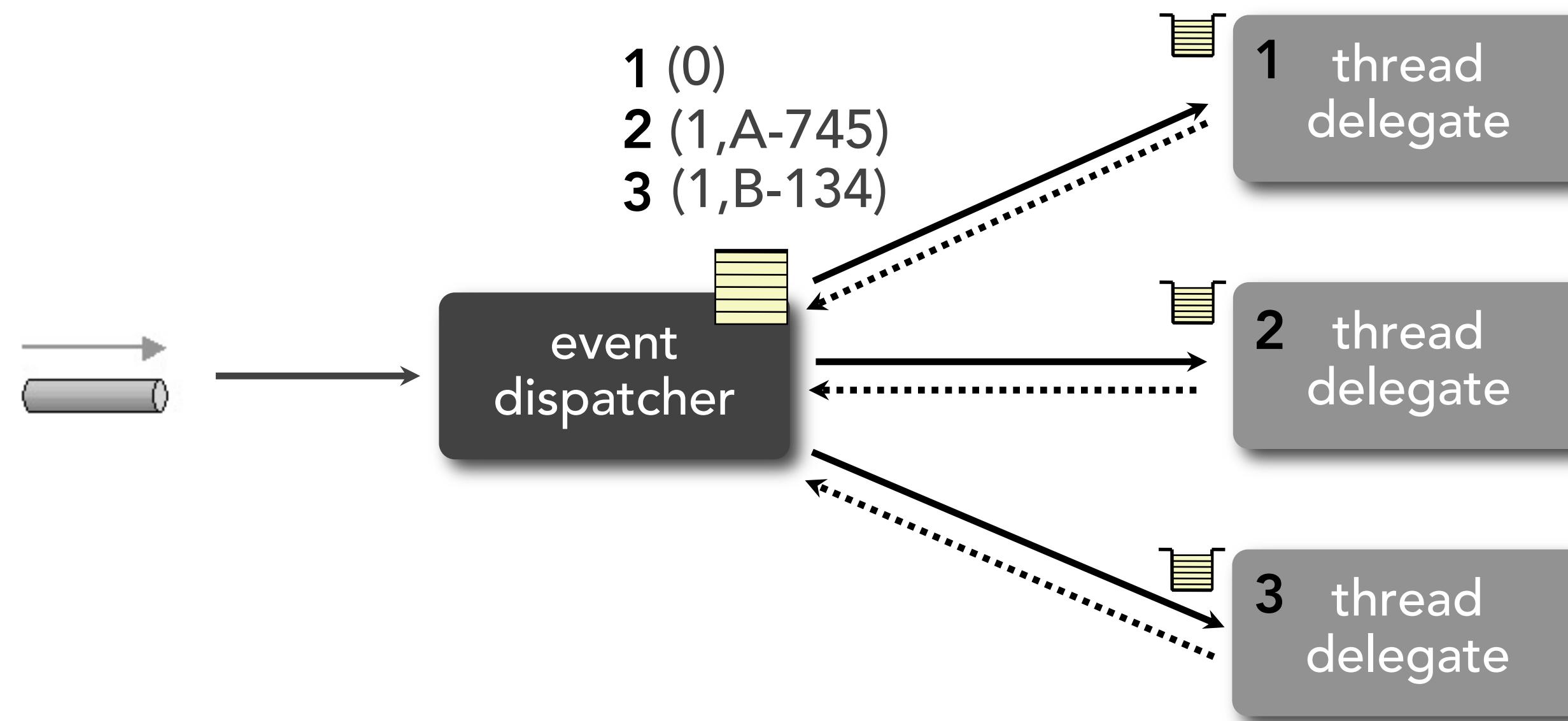
# thread delegate pattern



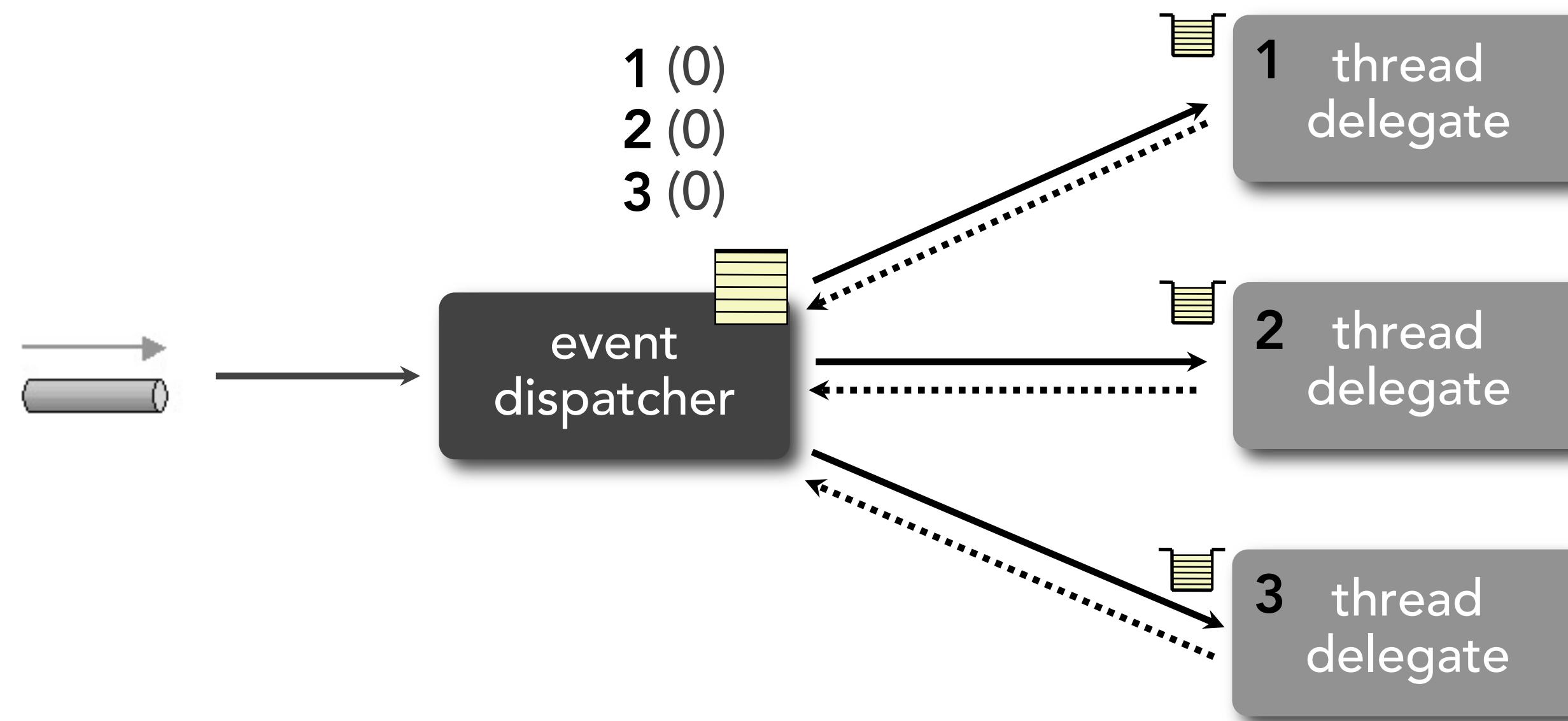
# thread delegate pattern



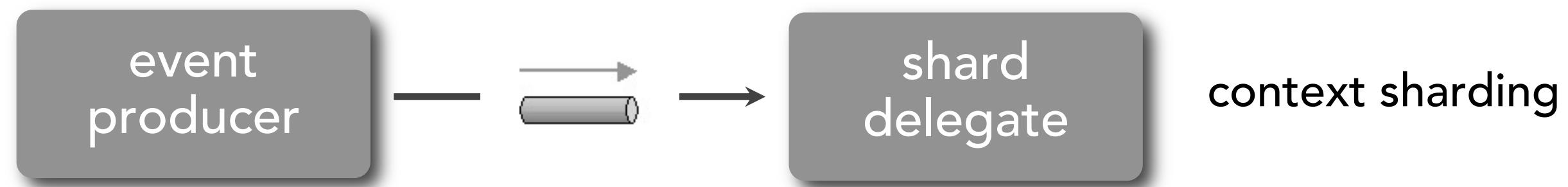
# thread delegate pattern



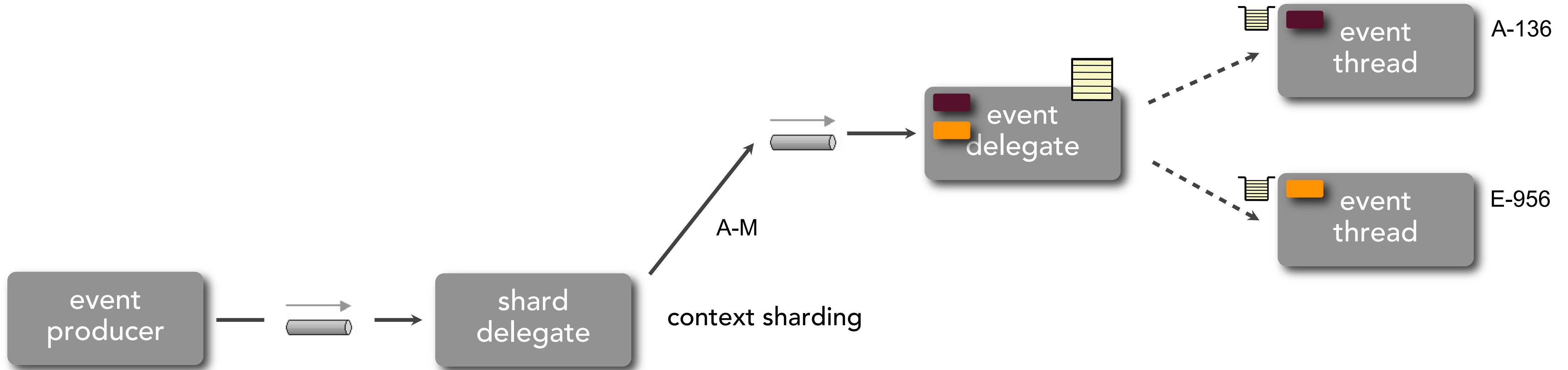
# thread delegate pattern



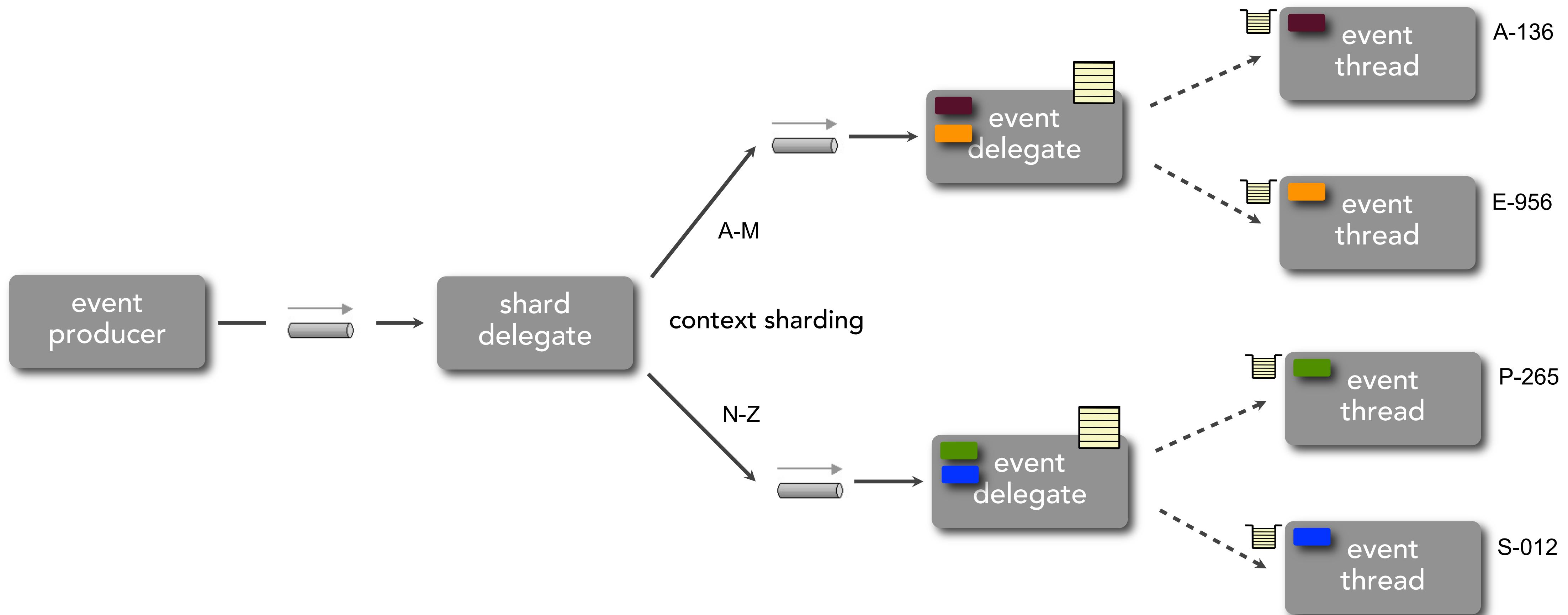
# thread delegate pattern



# thread delegate pattern



# thread delegate pattern

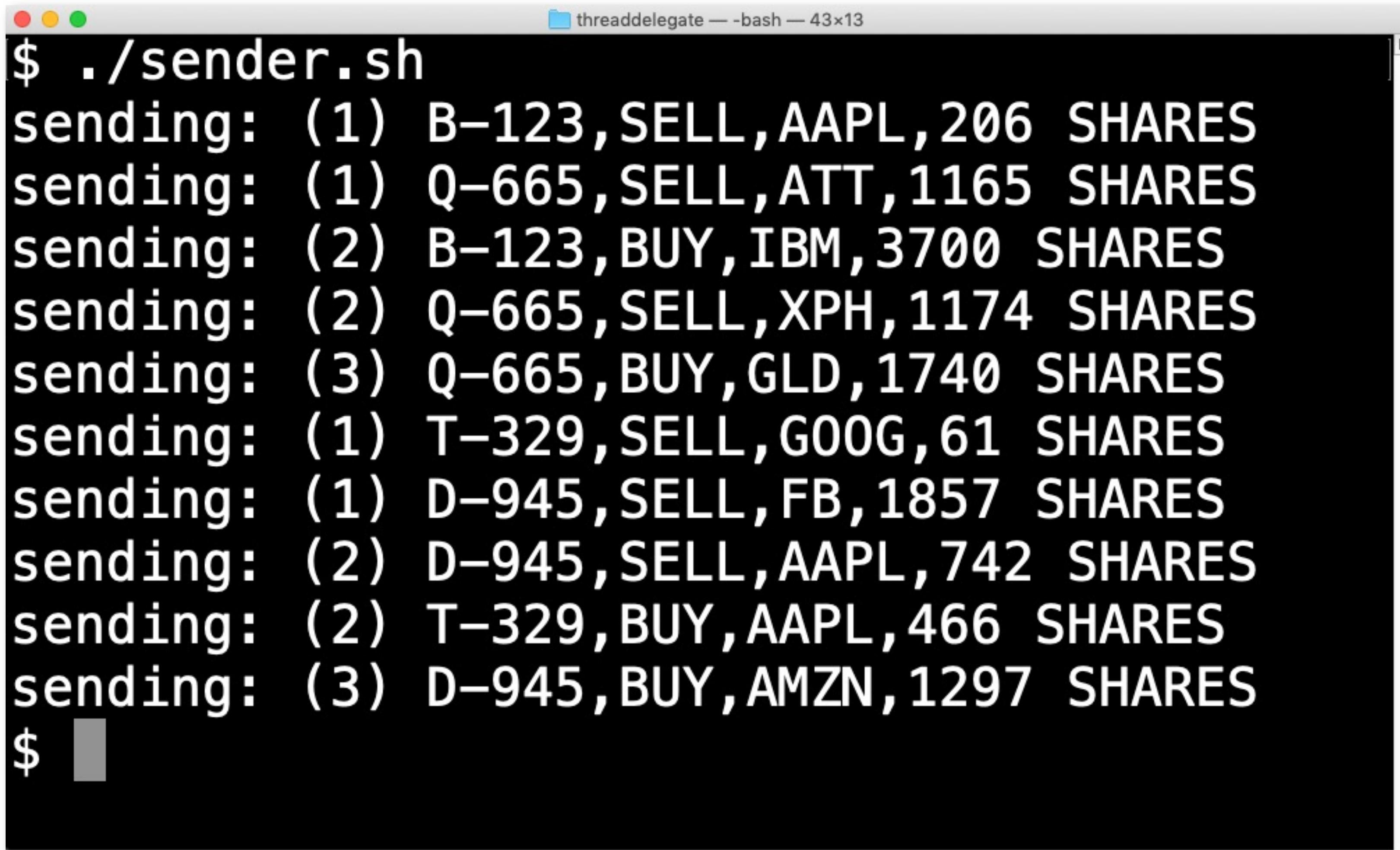


# thread delegate pattern



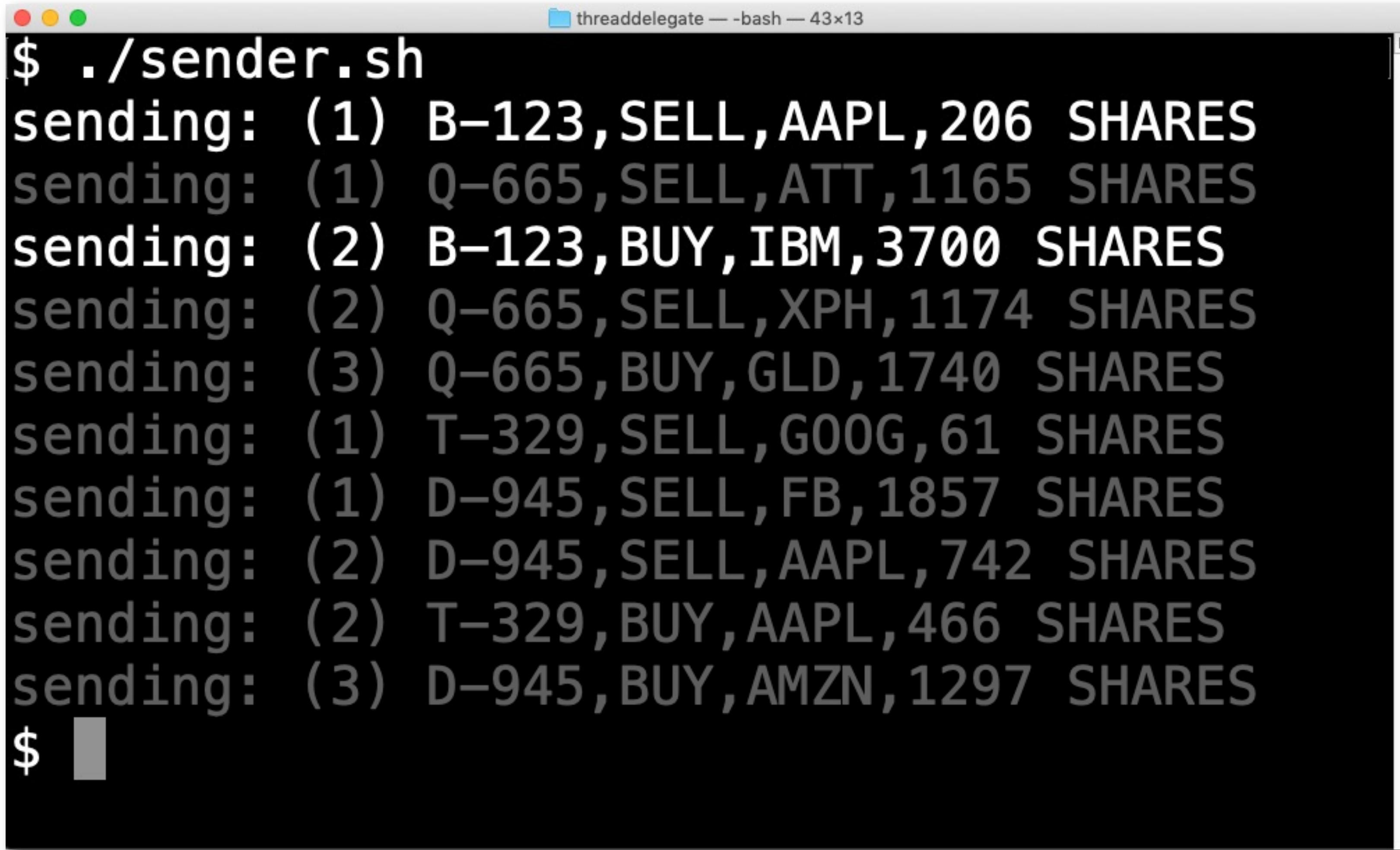
let's apply the pattern...

# thread delegate pattern



```
$ ./sender.sh
sending: (1) B-123,SELL,AAPL,206 SHARES
sending: (1) Q-665,SELL,ATT,1165 SHARES
sending: (2) B-123,BUY,IBM,3700 SHARES
sending: (2) Q-665,SELL,XPH,1174 SHARES
sending: (3) Q-665,BUY,GLD,1740 SHARES
sending: (1) T-329,SELL,GOOG,61 SHARES
sending: (1) D-945,SELL,FB,1857 SHARES
sending: (2) D-945,SELL,AAPL,742 SHARES
sending: (2) T-329,BUY,AAPL,466 SHARES
sending: (3) D-945,BUY,AMZN,1297 SHARES
$
```

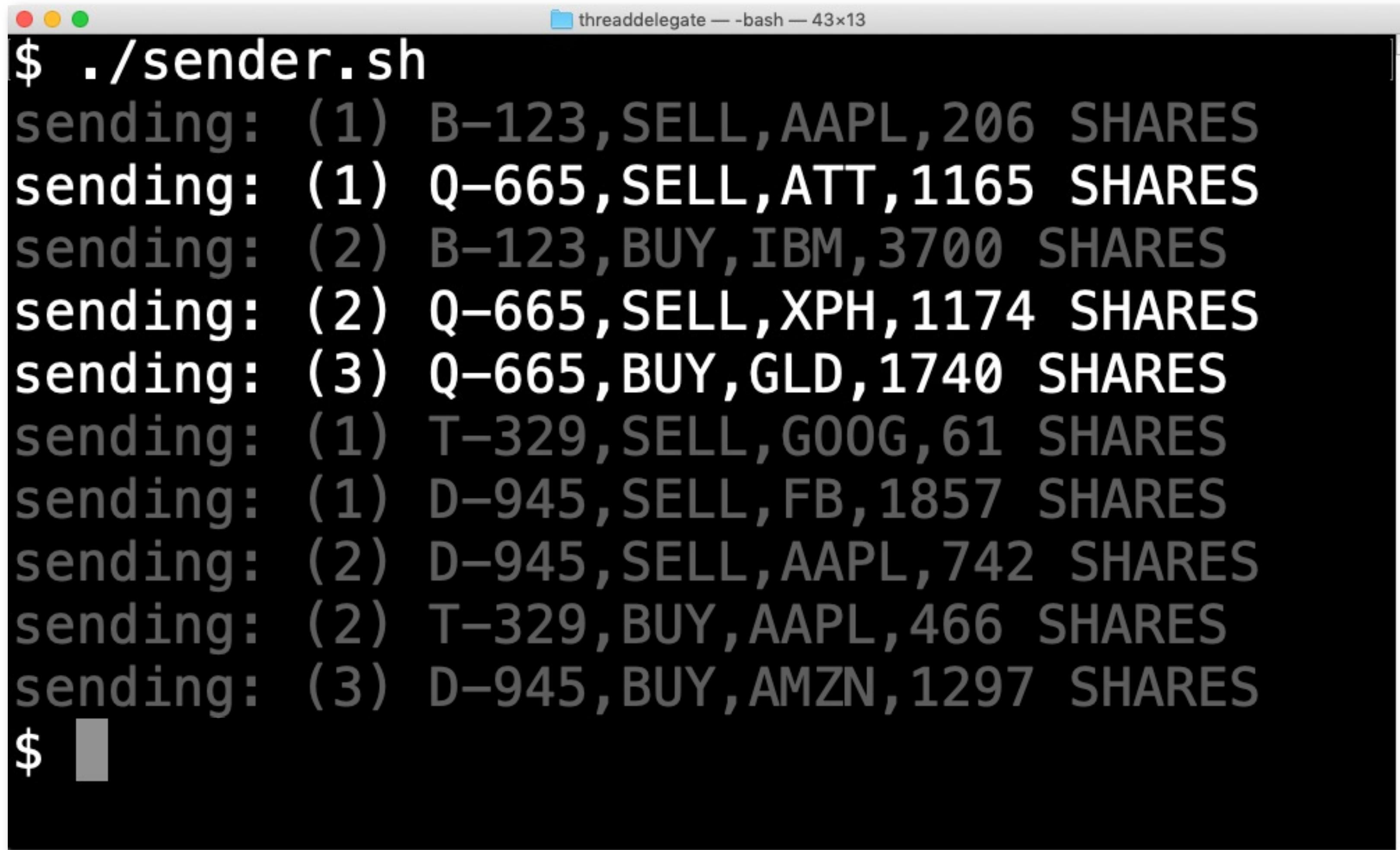
# thread delegate pattern



A terminal window titled "threaddelegate — bash — 43x13" displays the output of a shell script named "sender.sh". The script uses a thread delegate pattern to handle multiple tasks simultaneously. The output shows the script sending various financial transactions (orders) to a system. The transactions include sell orders for AAPL, ATT, XPH, and FB, and buy orders for IBM, GLD, GOOG, AAPL, and AMZN. The script sends three orders for each type of transaction.

```
$ ./sender.sh
sending: (1) B-123,SELL,AAPL,206 SHARES
sending: (1) Q-665,SELL,ATT,1165 SHARES
sending: (2) B-123,BUY,IBM,3700 SHARES
sending: (2) Q-665,SELL,XPH,1174 SHARES
sending: (3) Q-665,BUY,GLD,1740 SHARES
sending: (1) T-329,SELL,GOOG,61 SHARES
sending: (1) D-945,SELL,FB,1857 SHARES
sending: (2) D-945,SELL,AAPL,742 SHARES
sending: (2) T-329,BUY,AAPL,466 SHARES
sending: (3) D-945,BUY,AMZN,1297 SHARES
$ █
```

# thread delegate pattern

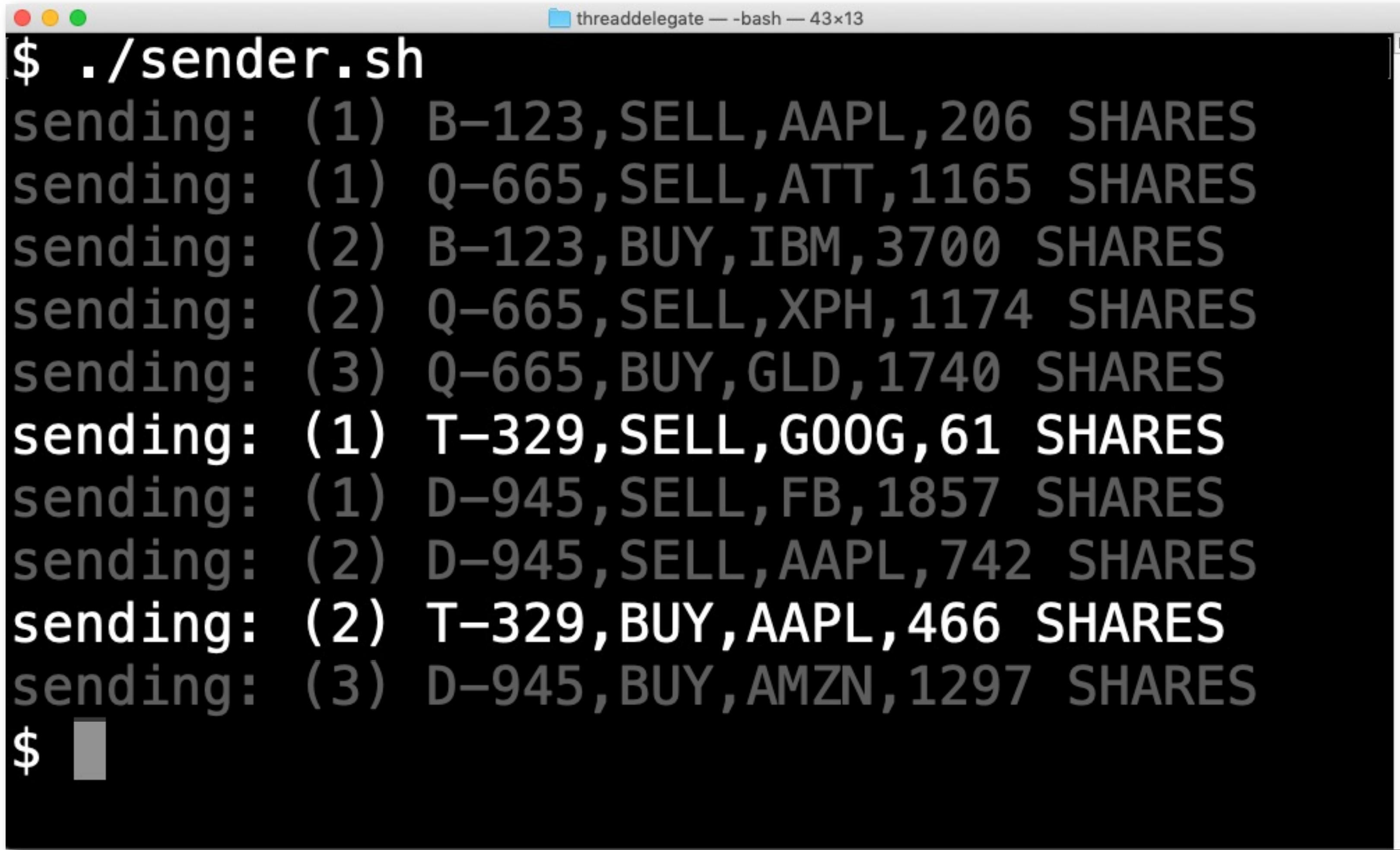


A terminal window titled "threaddelegate — bash — 43x13" is shown. The window contains the command \$ ./sender.sh followed by ten lines of output. Each line starts with the word "sending:" followed by a transaction record in parentheses. The records are:

- (1) B-123,SELL,AAPL,206 SHARES
- (1) Q-665,SELL,ATT,1165 SHARES
- (2) B-123,BUY,IBM,3700 SHARES
- (2) Q-665,SELL,XPH,1174 SHARES
- (3) Q-665,BUY,GLD,1740 SHARES
- (1) T-329,SELL,GOOG,61 SHARES
- (1) D-945,SELL,FB,1857 SHARES
- (2) D-945,SELL,AAPL,742 SHARES
- (2) T-329,BUY,AAPL,466 SHARES
- (3) D-945,BUY,AMZN,1297 SHARES

The prompt \$ is visible at the bottom left.

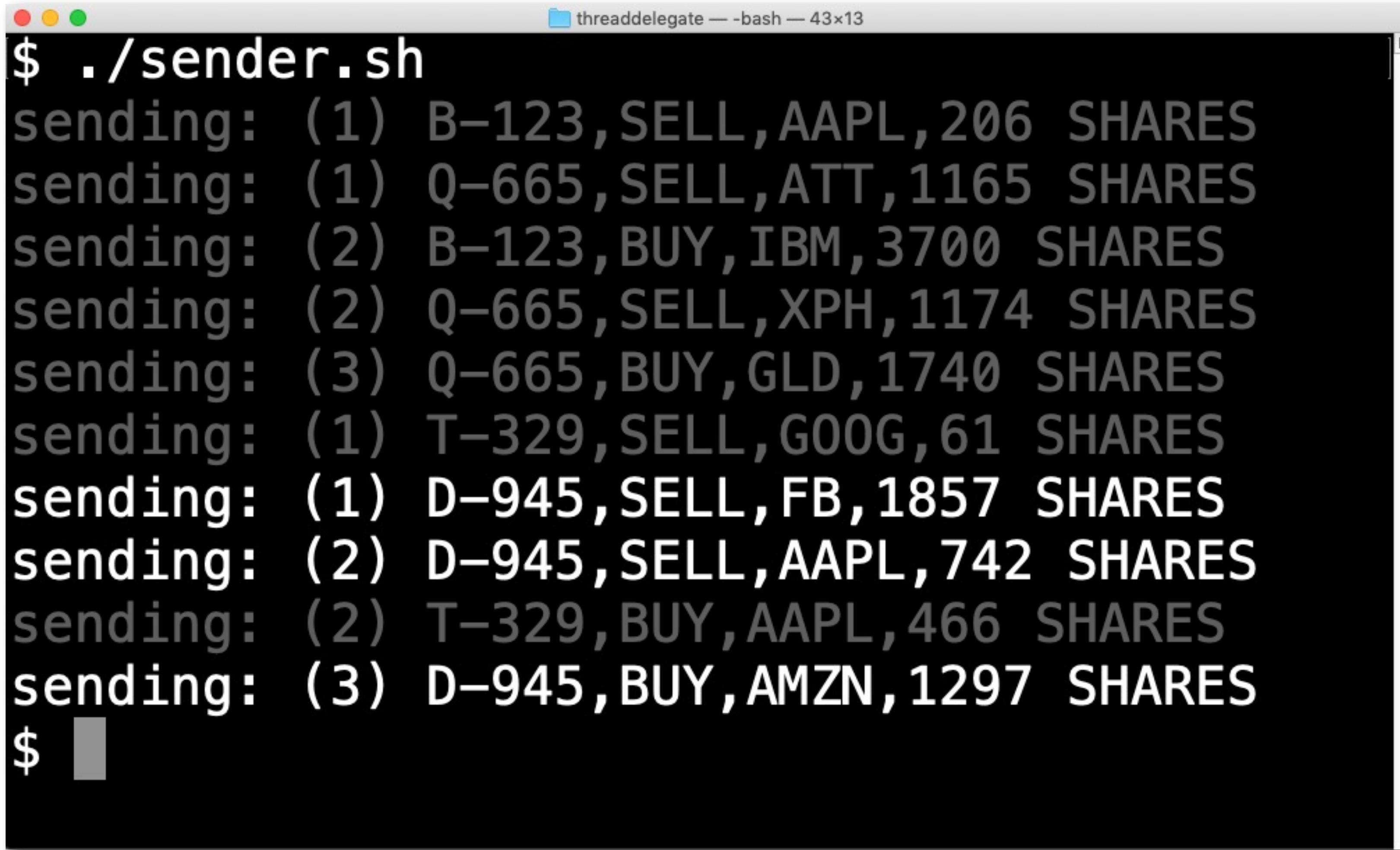
# thread delegate pattern



A terminal window titled "threaddelegate — bash — 43x13" is shown. The window contains the command \$ ./sender.sh followed by ten lines of output. Each line starts with the word "sending:" followed by a transaction identifier (1 through 5) and a trade detail (e.g., B-123, SELL, AAPL, 206 SHARES). The terminal has a dark background with light-colored text.

```
$ ./sender.sh
sending: (1) B-123,SELL,AAPL,206 SHARES
sending: (1) Q-665,SELL,ATT,1165 SHARES
sending: (2) B-123,BUY,IBM,3700 SHARES
sending: (2) Q-665,SELL,XPH,1174 SHARES
sending: (3) Q-665,BUY,GLD,1740 SHARES
sending: (1) T-329,SELL,GOOG,61 SHARES
sending: (1) D-945,SELL,FB,1857 SHARES
sending: (2) D-945,SELL,AAPL,742 SHARES
sending: (2) T-329,BUY,AAPL,466 SHARES
sending: (3) D-945,BUY,AMZN,1297 SHARES
$ █
```

# thread delegate pattern

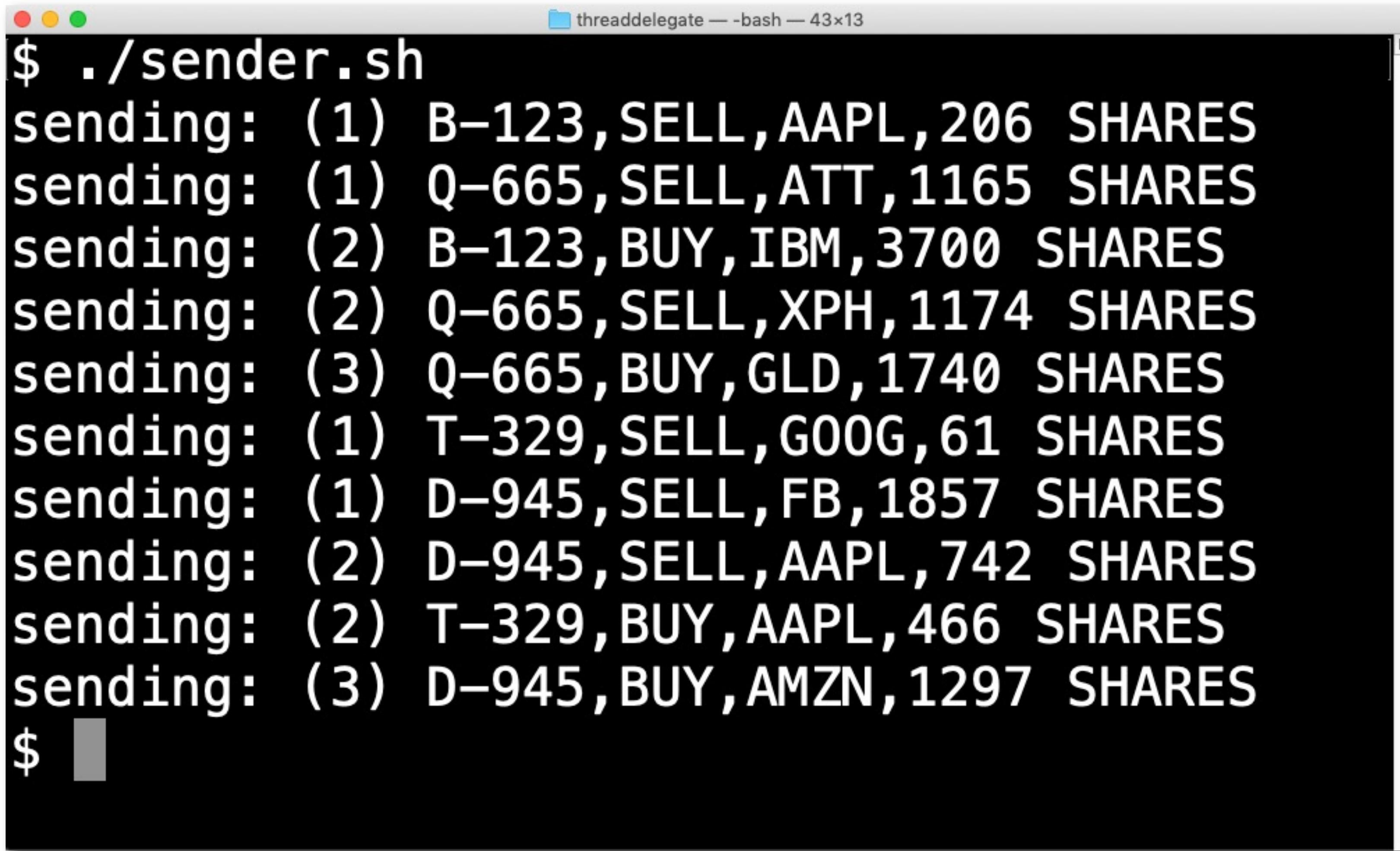


A terminal window titled "threaddelegate — bash — 43x13" is shown. The window contains the command \$ ./sender.sh followed by ten lines of output. Each line starts with the word "sending:" followed by a transaction record in parentheses. The records are:

- (1) B-123,SELL,AAPL,206 SHARES
- (1) Q-665,SELL,ATT,1165 SHARES
- (2) B-123,BUY,IBM,3700 SHARES
- (2) Q-665,SELL,XPH,1174 SHARES
- (3) Q-665,BUY,GLD,1740 SHARES
- (1) T-329,SELL,GOOG,61 SHARES
- (1) D-945,SELL,FB,1857 SHARES
- (2) D-945,SELL,AAPL,742 SHARES
- (2) T-329,BUY,AAPL,466 SHARES
- (3) D-945,BUY,AMZN,1297 SHARES

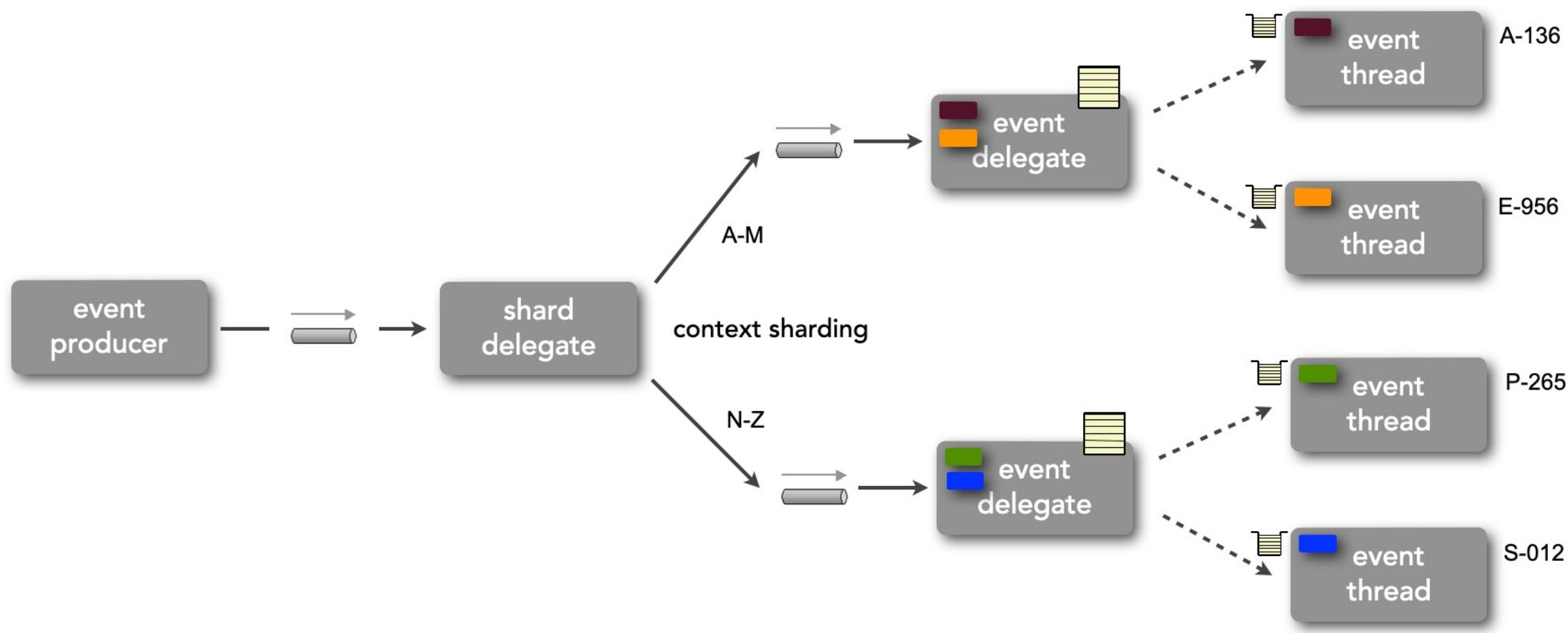
The terminal prompt \$ is visible at the bottom left.

# thread delegate pattern

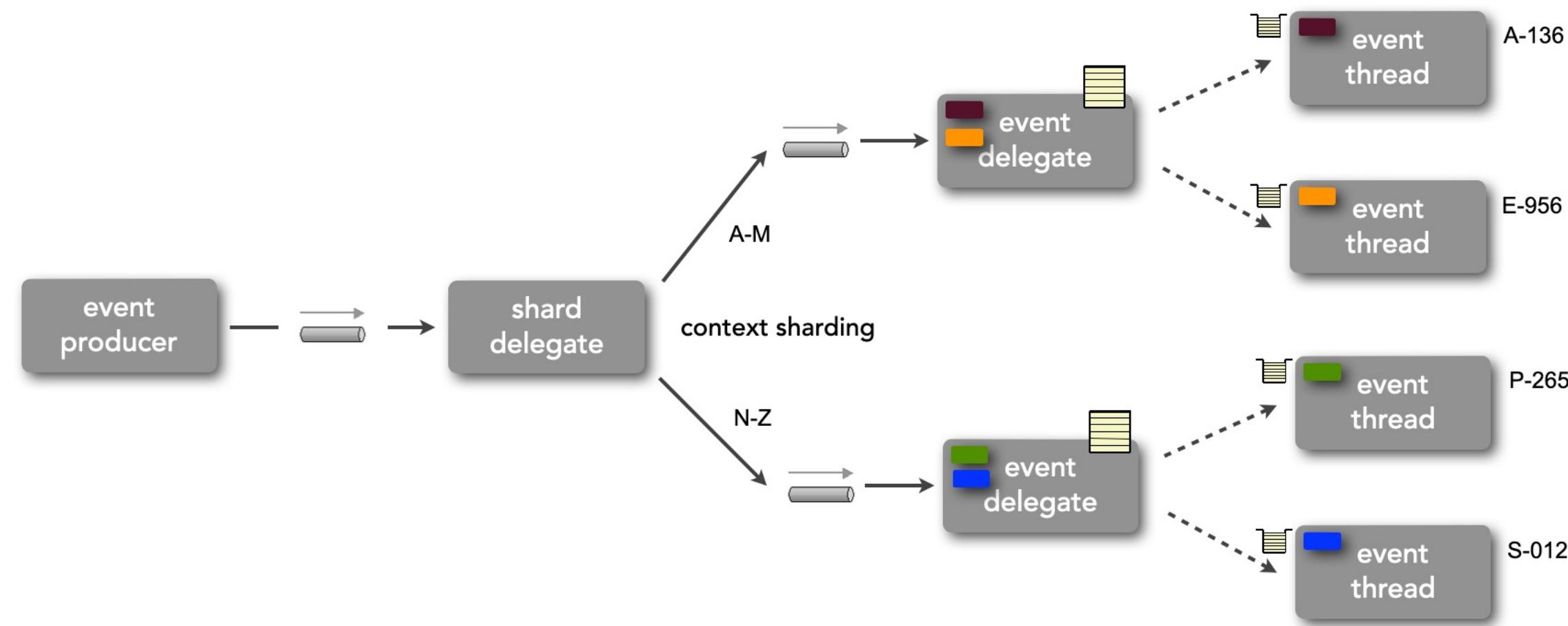


```
$ ./sender.sh
sending: (1) B-123,SELL,AAPL,206 SHARES
sending: (1) Q-665,SELL,ATT,1165 SHARES
sending: (2) B-123,BUY,IBM,3700 SHARES
sending: (2) Q-665,SELL,XPH,1174 SHARES
sending: (3) Q-665,BUY,GLD,1740 SHARES
sending: (1) T-329,SELL,GOOG,61 SHARES
sending: (1) D-945,SELL,FB,1857 SHARES
sending: (2) D-945,SELL,AAPL,742 SHARES
sending: (2) T-329,BUY,AAPL,466 SHARES
sending: (3) D-945,BUY,AMZN,1297 SHARES
$
```

# thread delegate pattern



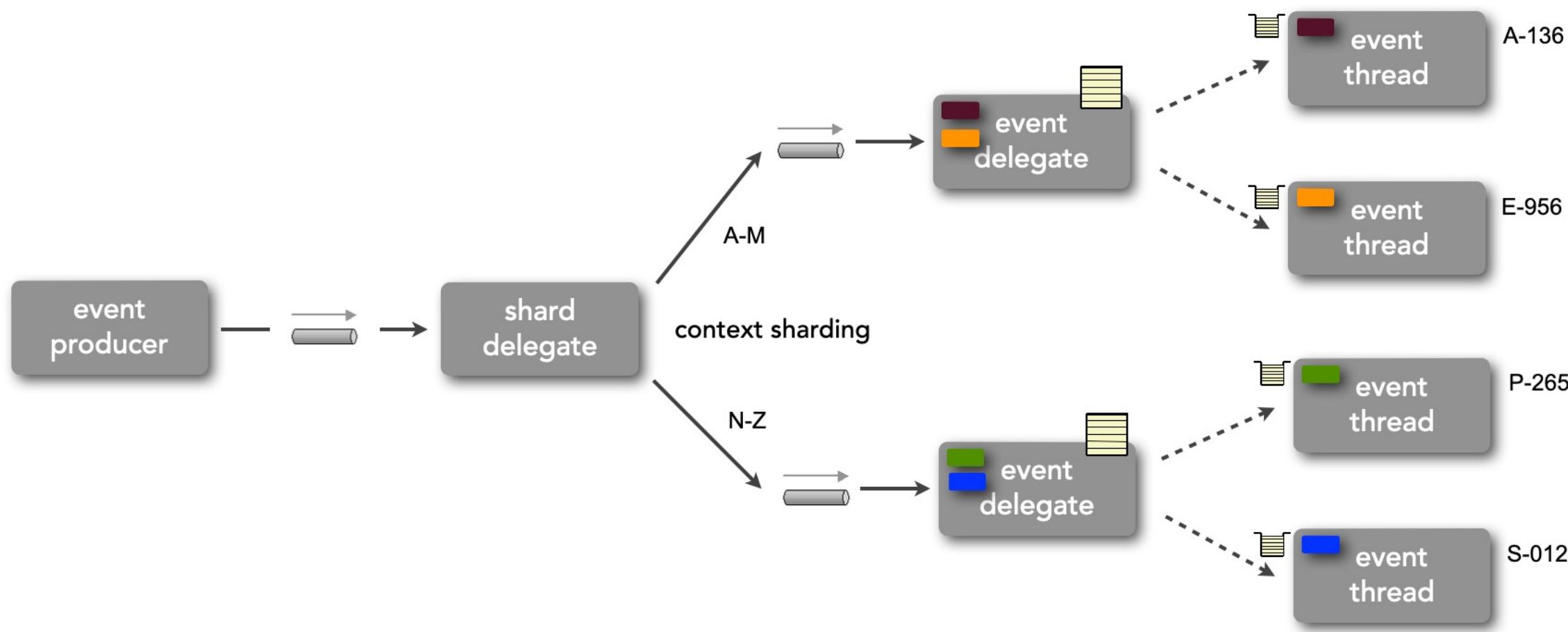
# thread delegate pattern



maintain processing order  
increase throughput  
increase performance  
increase scalability



# thread delegate pattern



- maintain processing order
- increase throughput
- increase performance
- increase scalability



- increased complexity
- possible thread saturation
- complex error handling
- increased cost

# Ambulance Pattern (Carpool)

# ambulance pattern



<https://github.com/wmr513/event-driven-patterns/tree/master/ambulance>

# ambulance pattern

*“how can I give some events a higher priority and still maintain responsiveness for all other events?”*

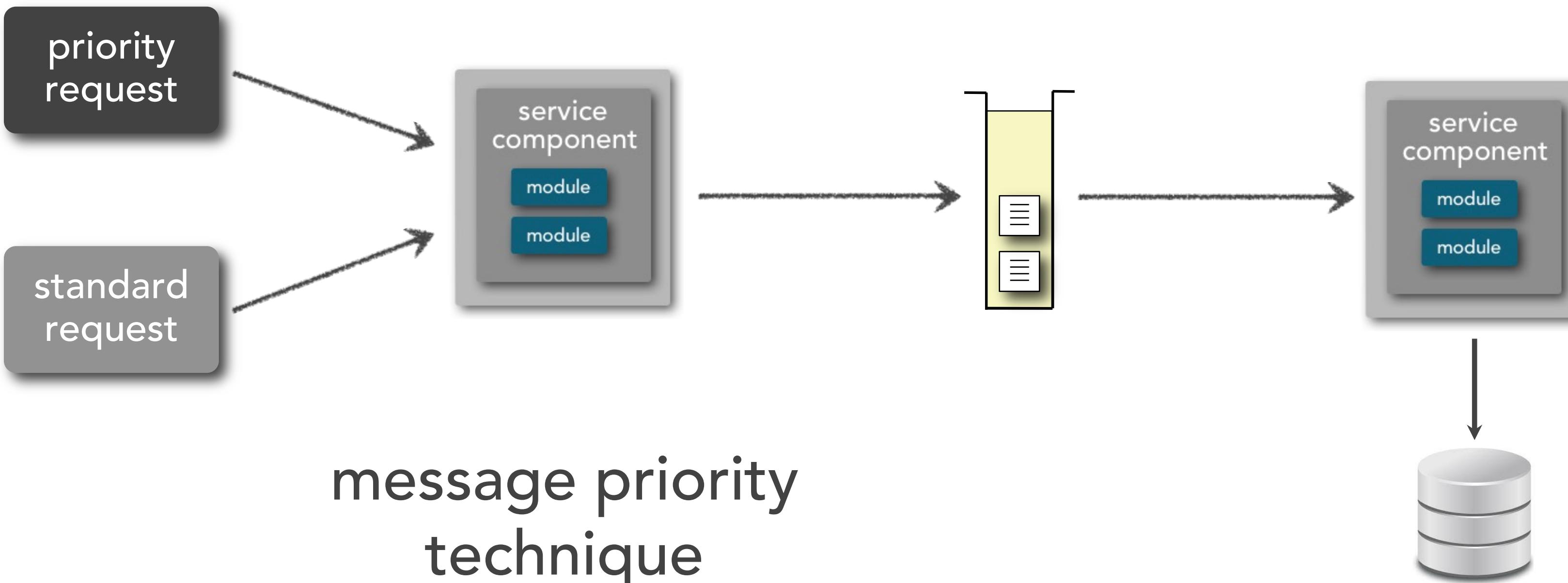


# ambulance pattern

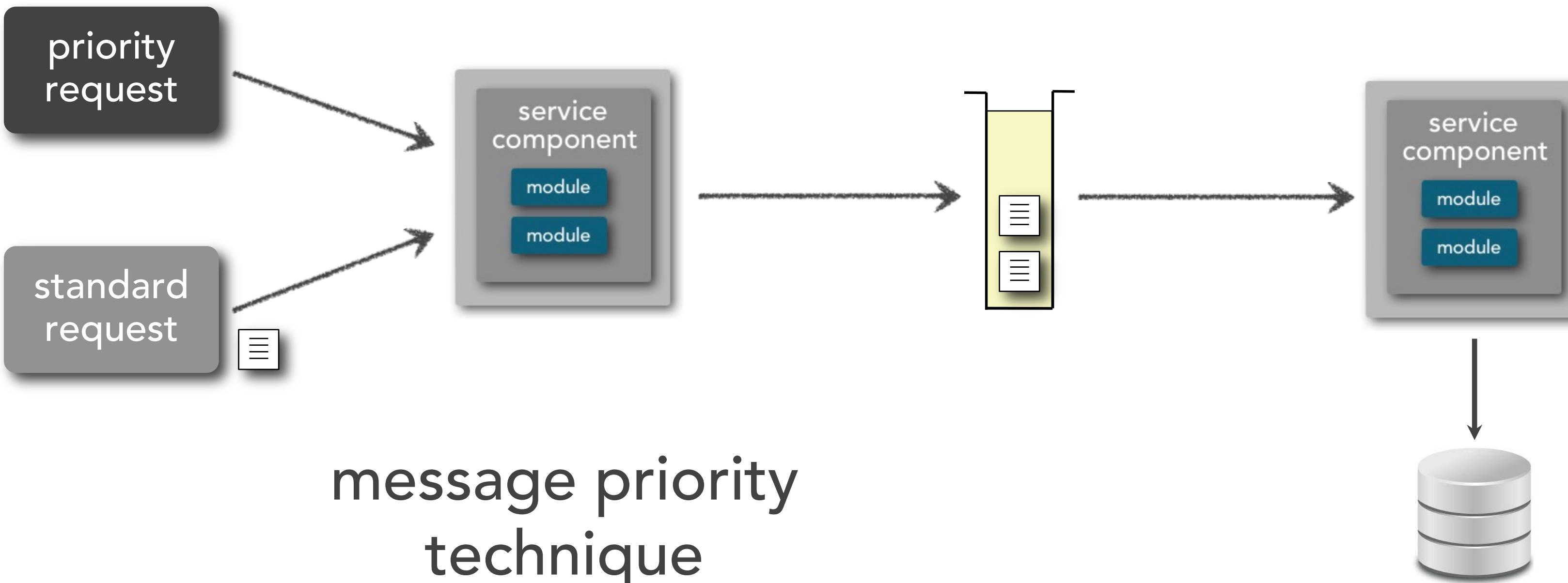


let's see the issue...

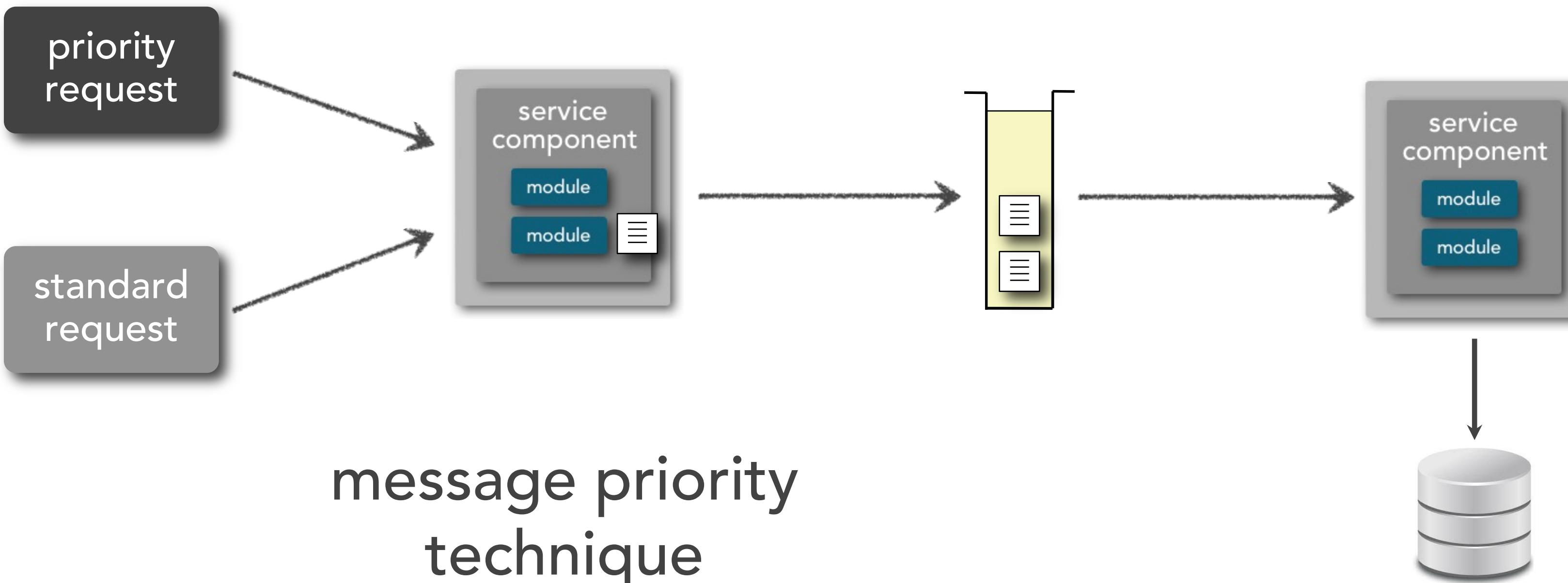
# ambulance pattern



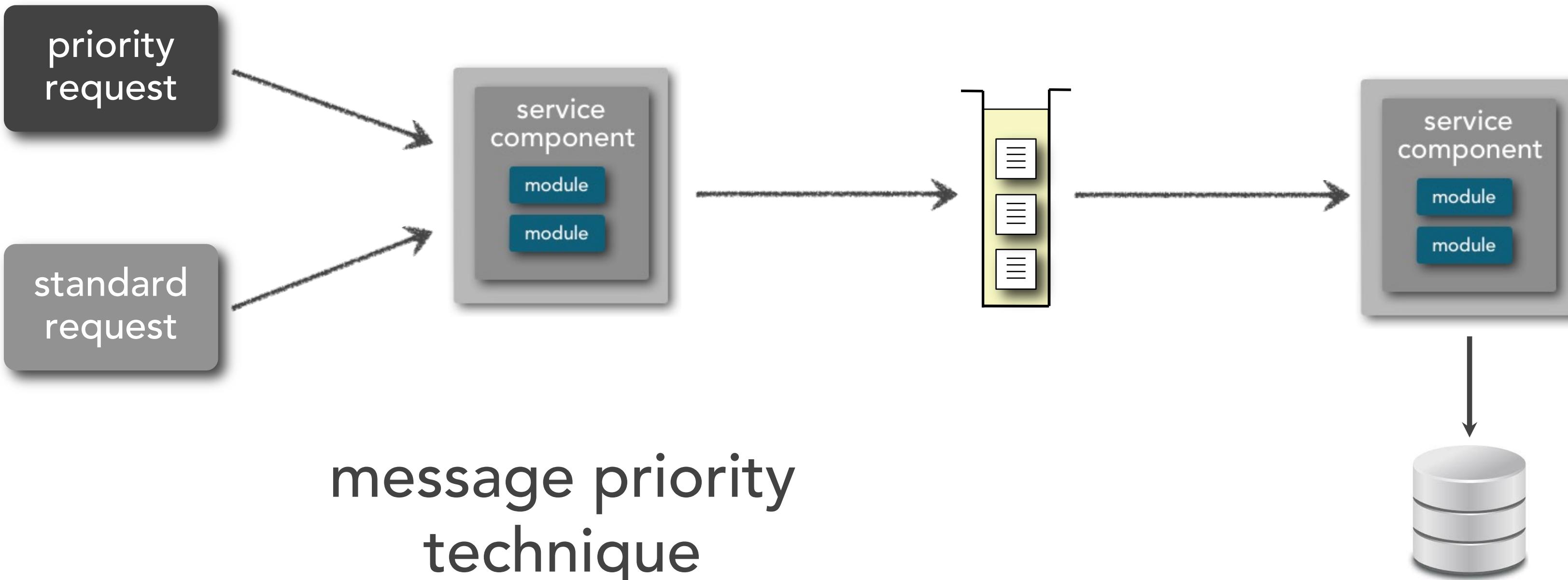
# ambulance pattern



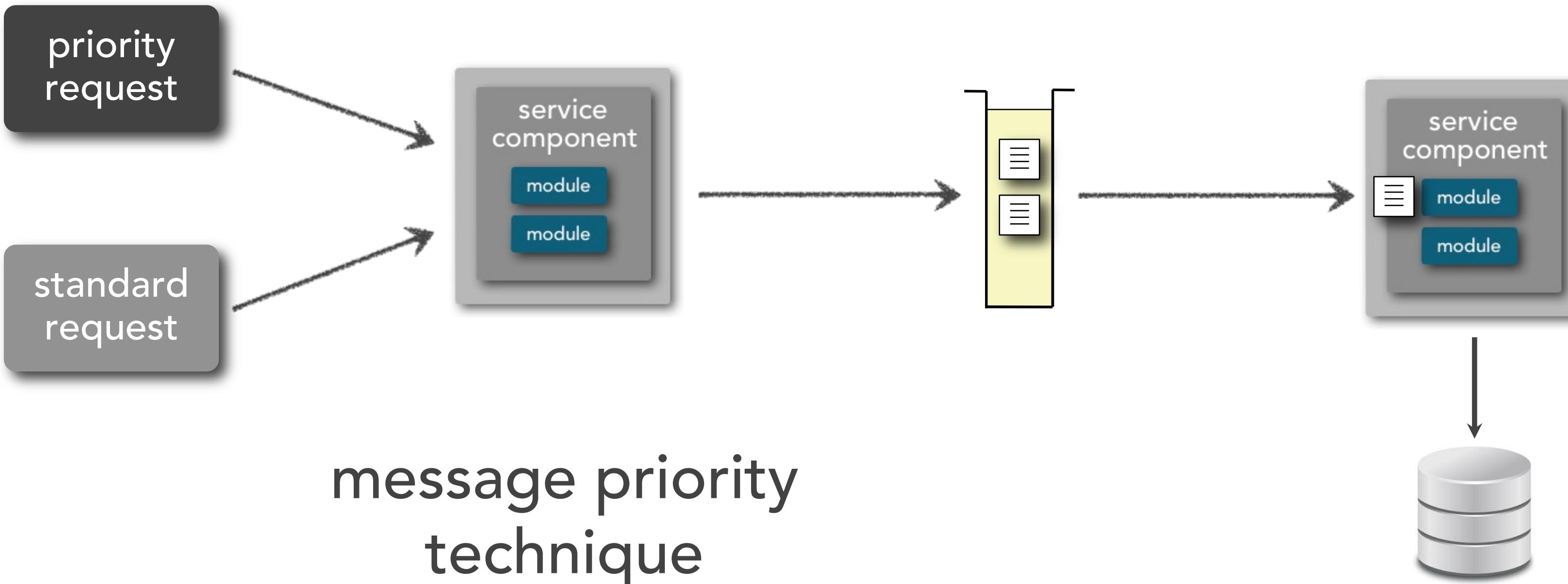
# ambulance pattern



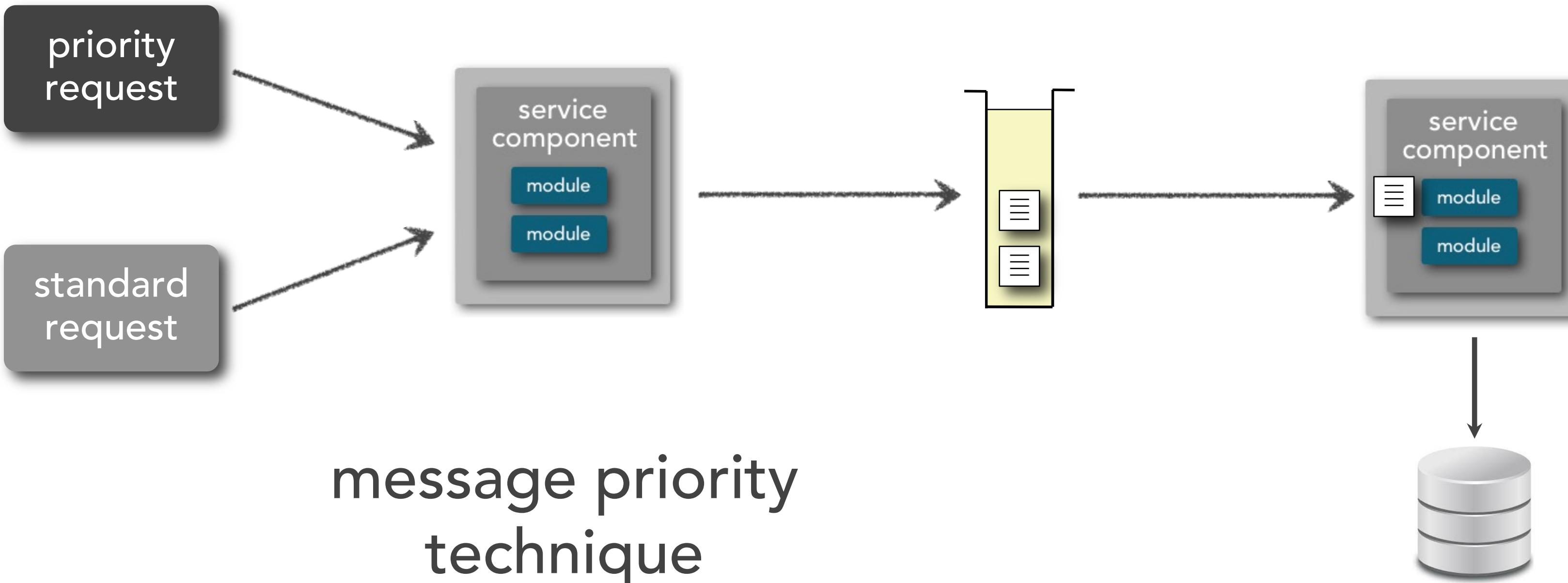
# ambulance pattern



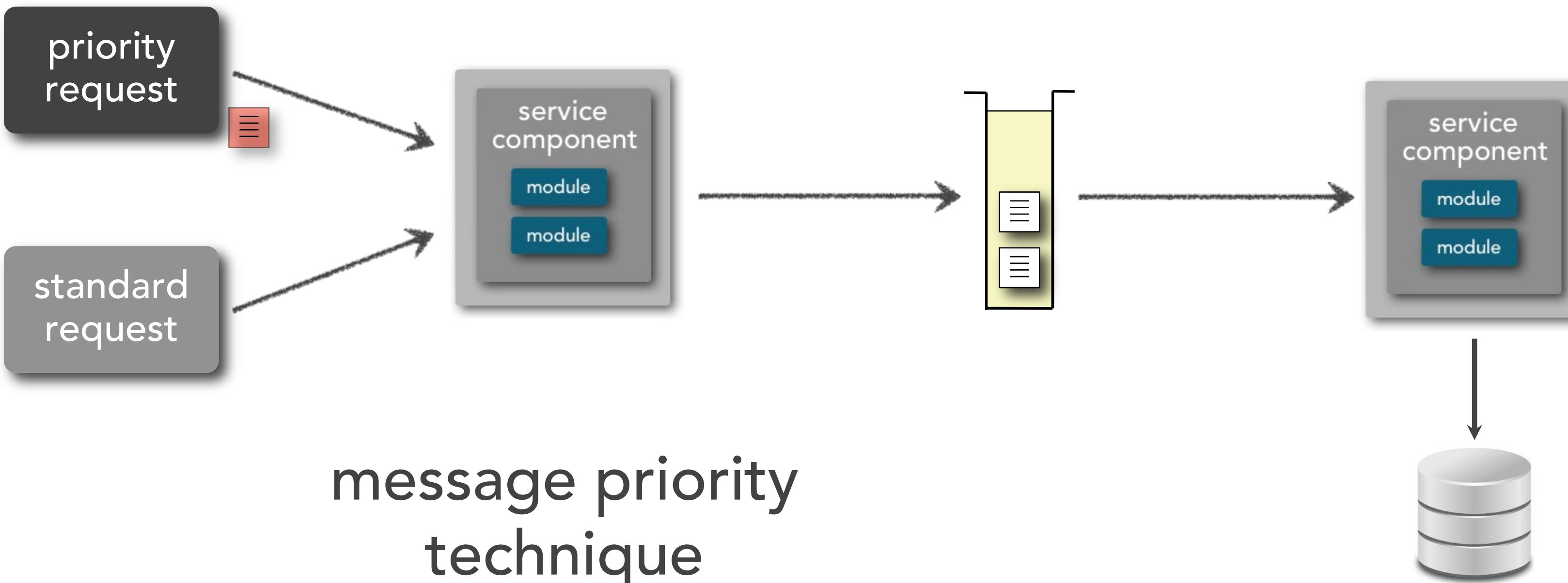
# ambulance pattern



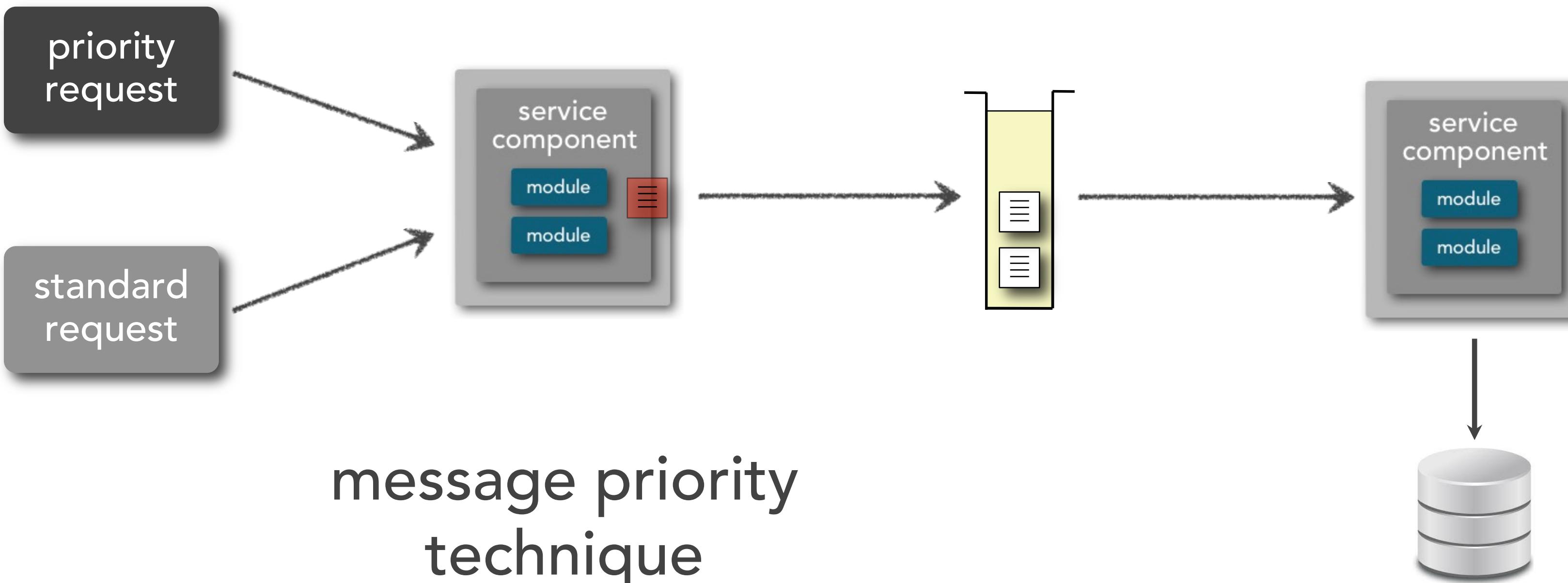
# ambulance pattern



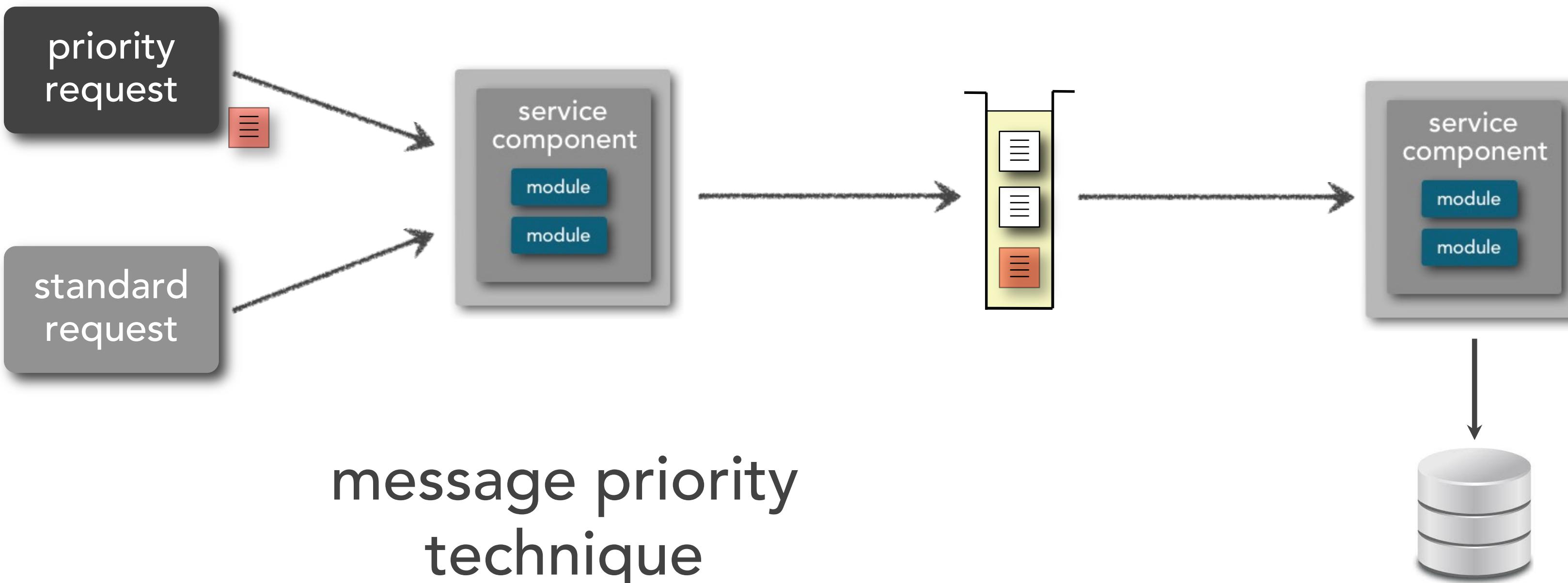
# ambulance pattern



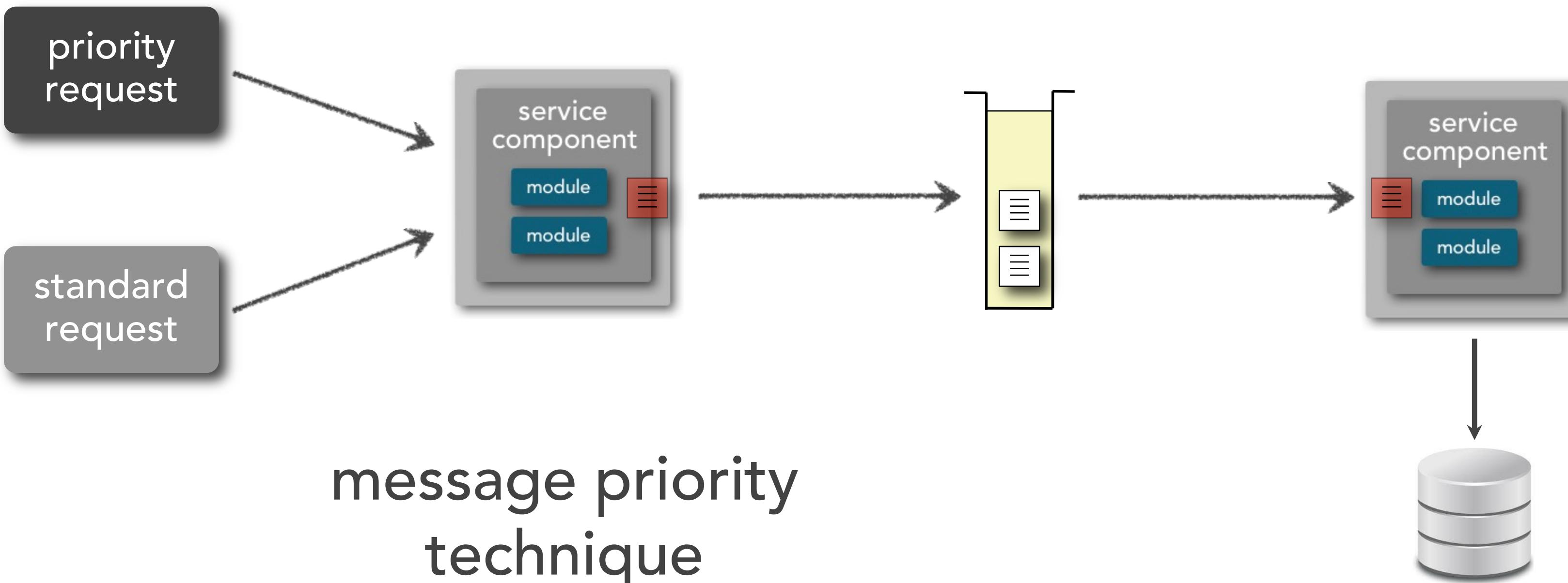
# ambulance pattern



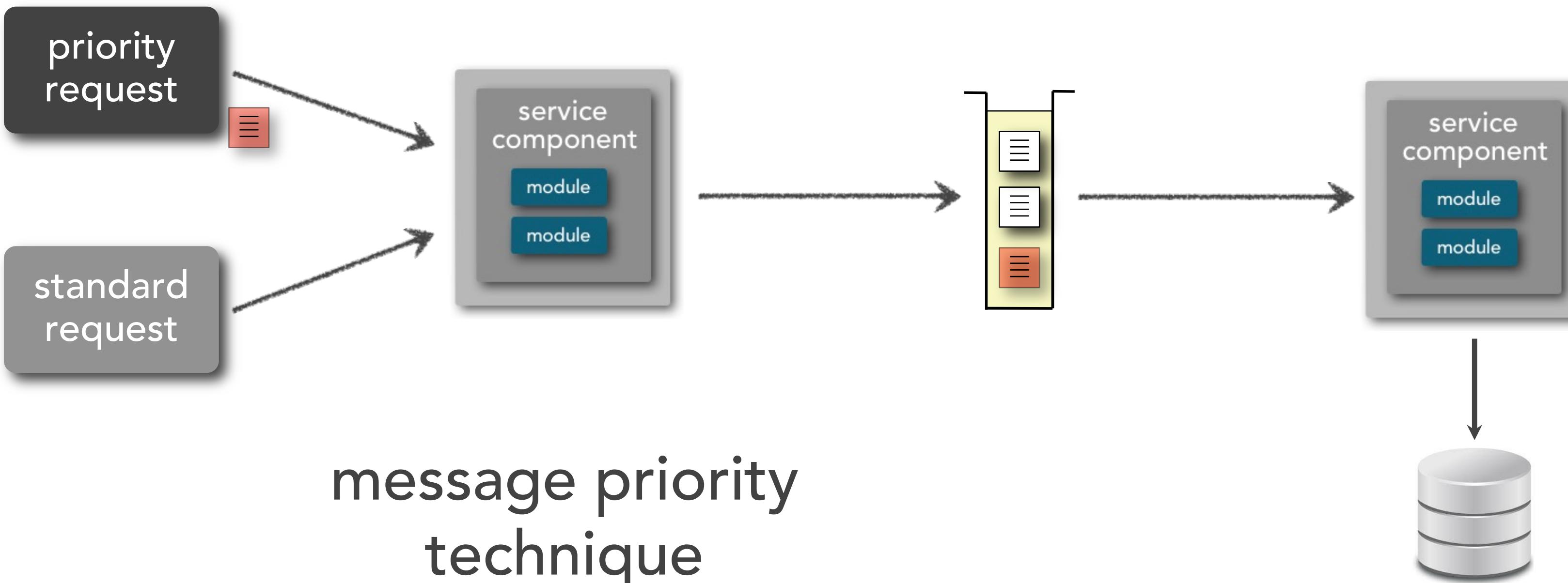
# ambulance pattern



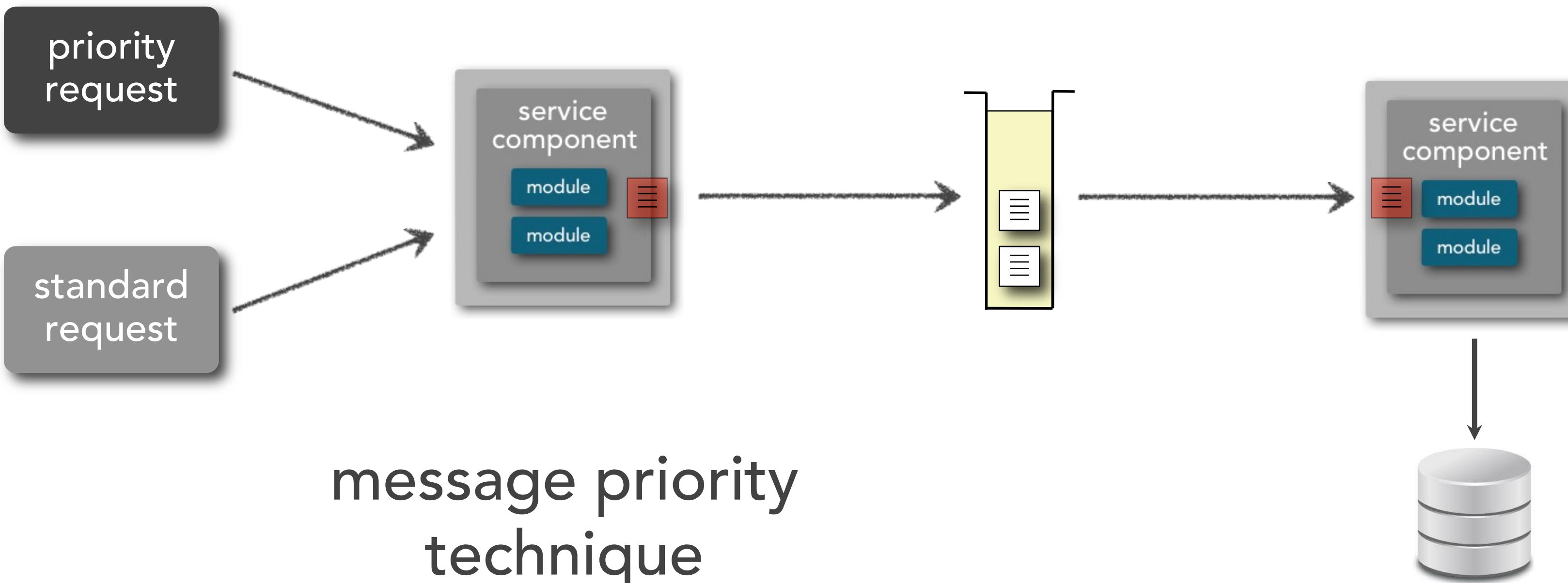
# ambulance pattern



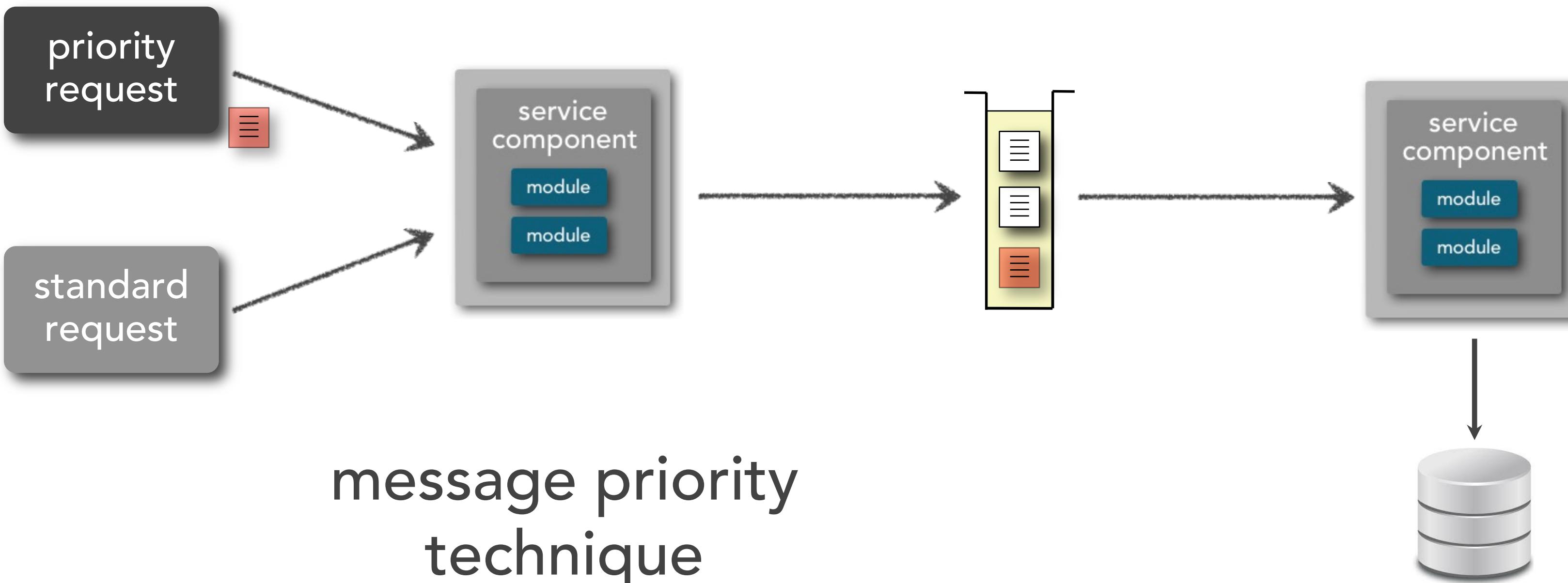
# ambulance pattern



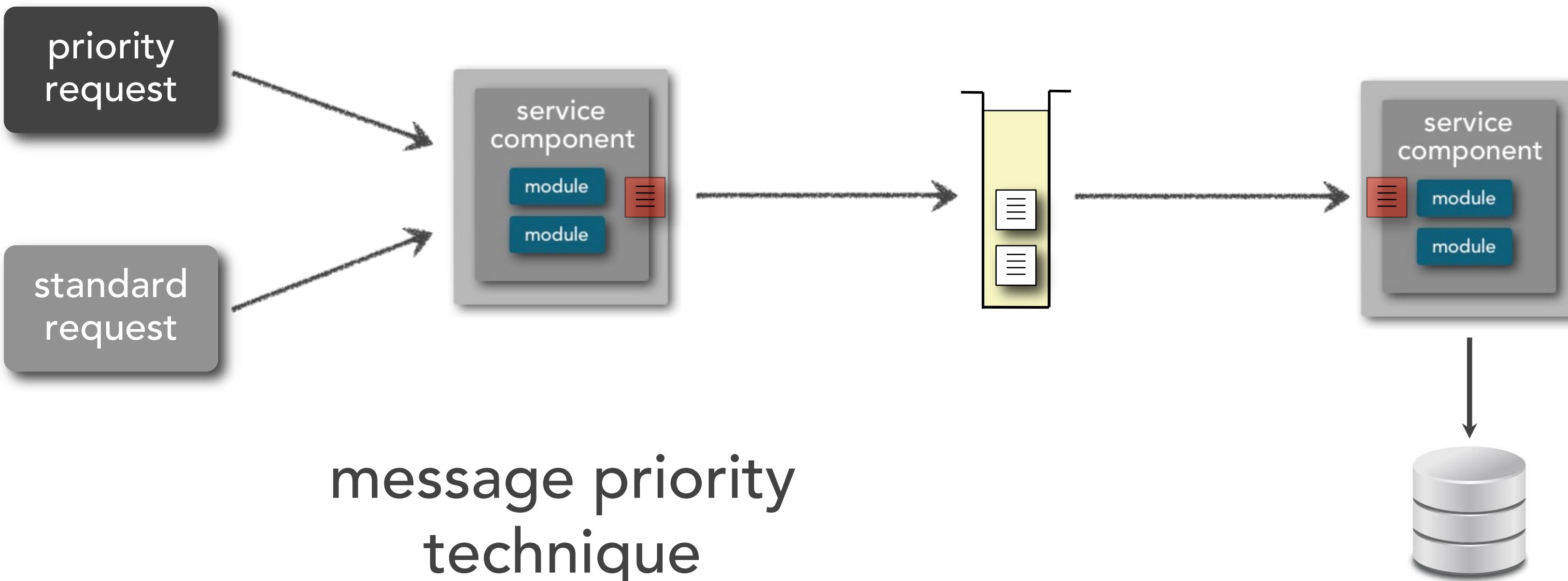
# ambulance pattern



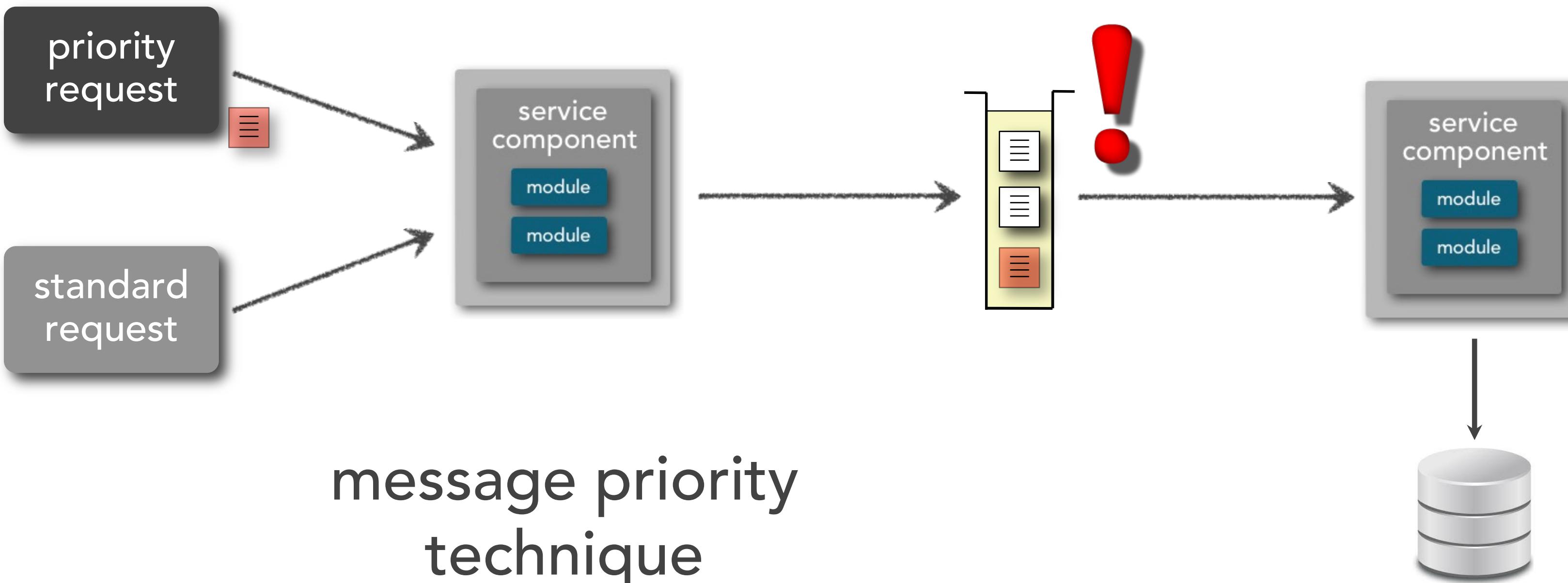
# ambulance pattern



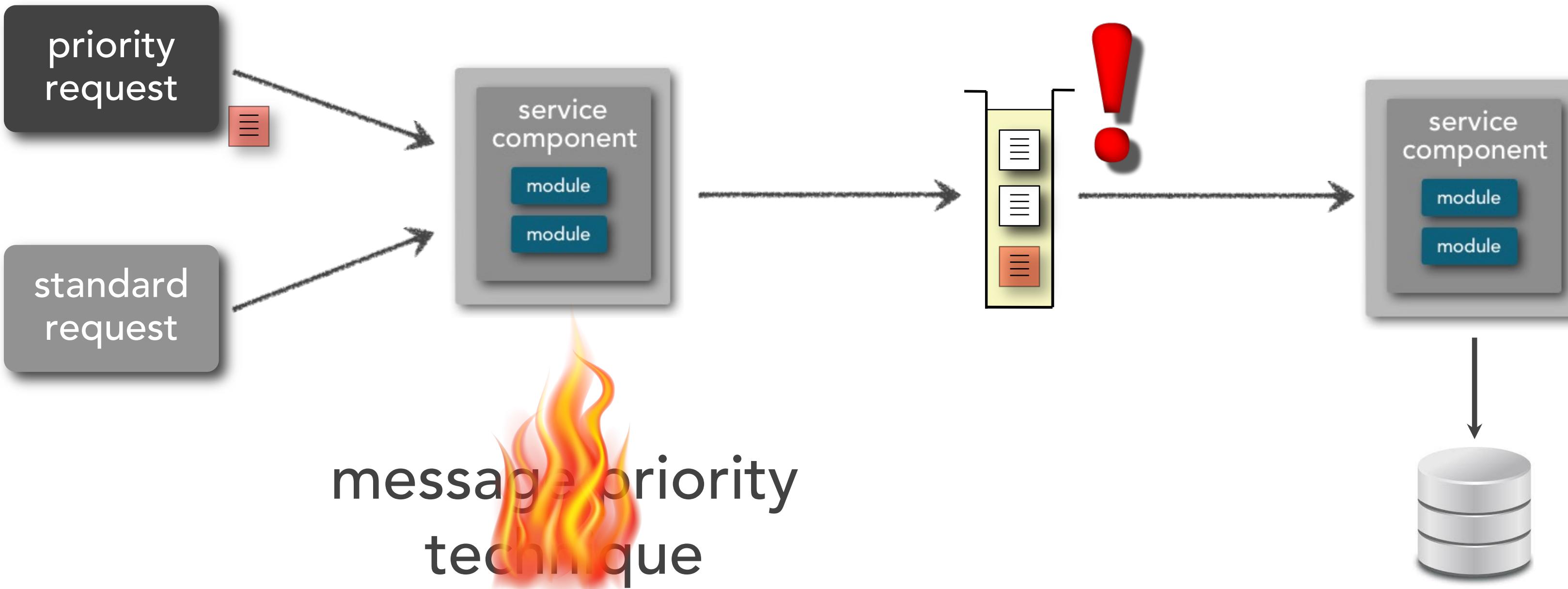
# ambulance pattern



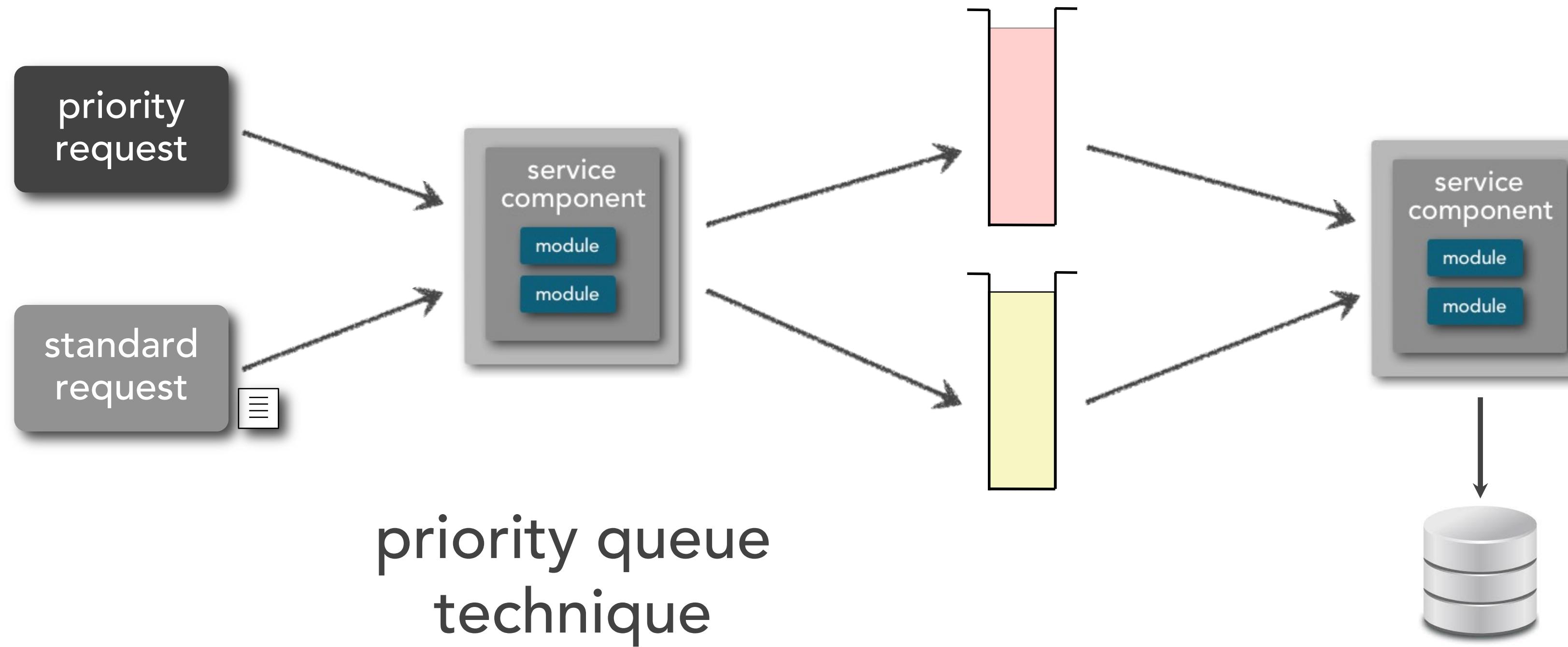
# ambulance pattern



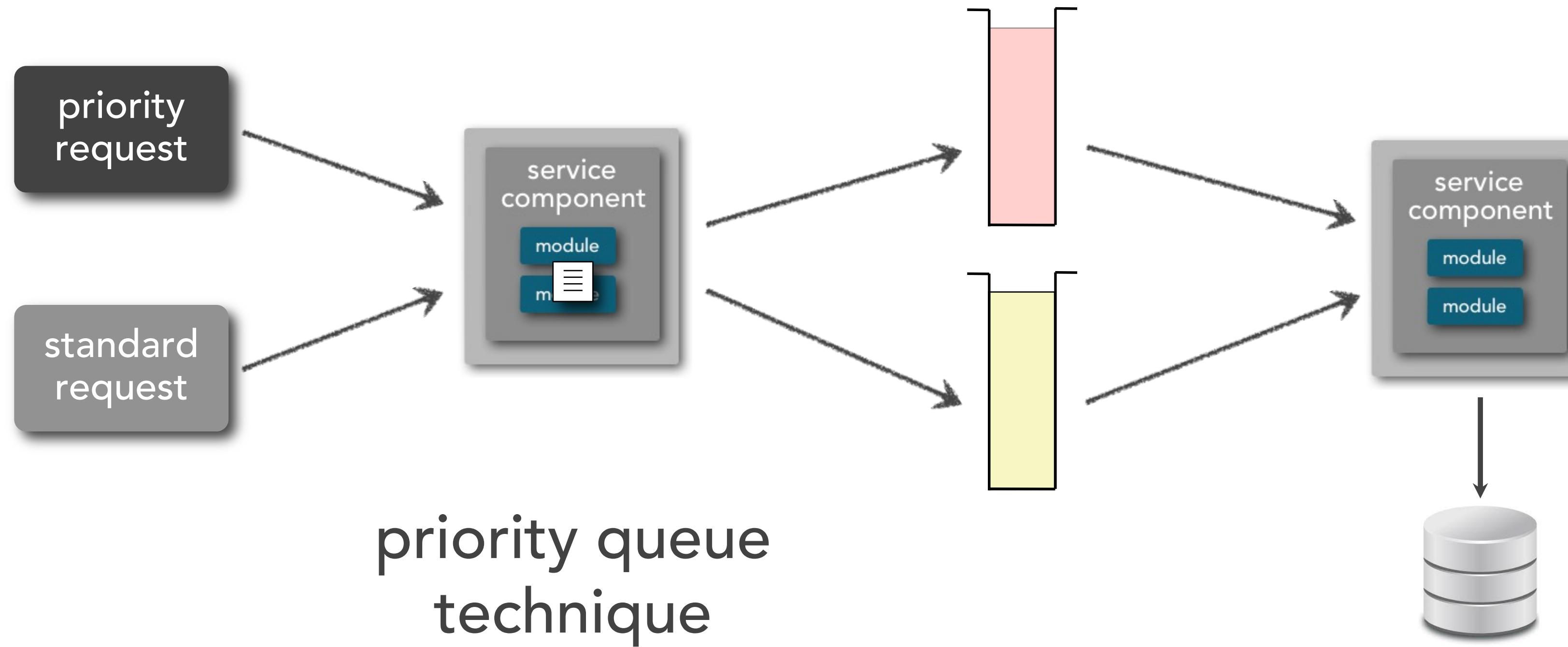
# ambulance pattern



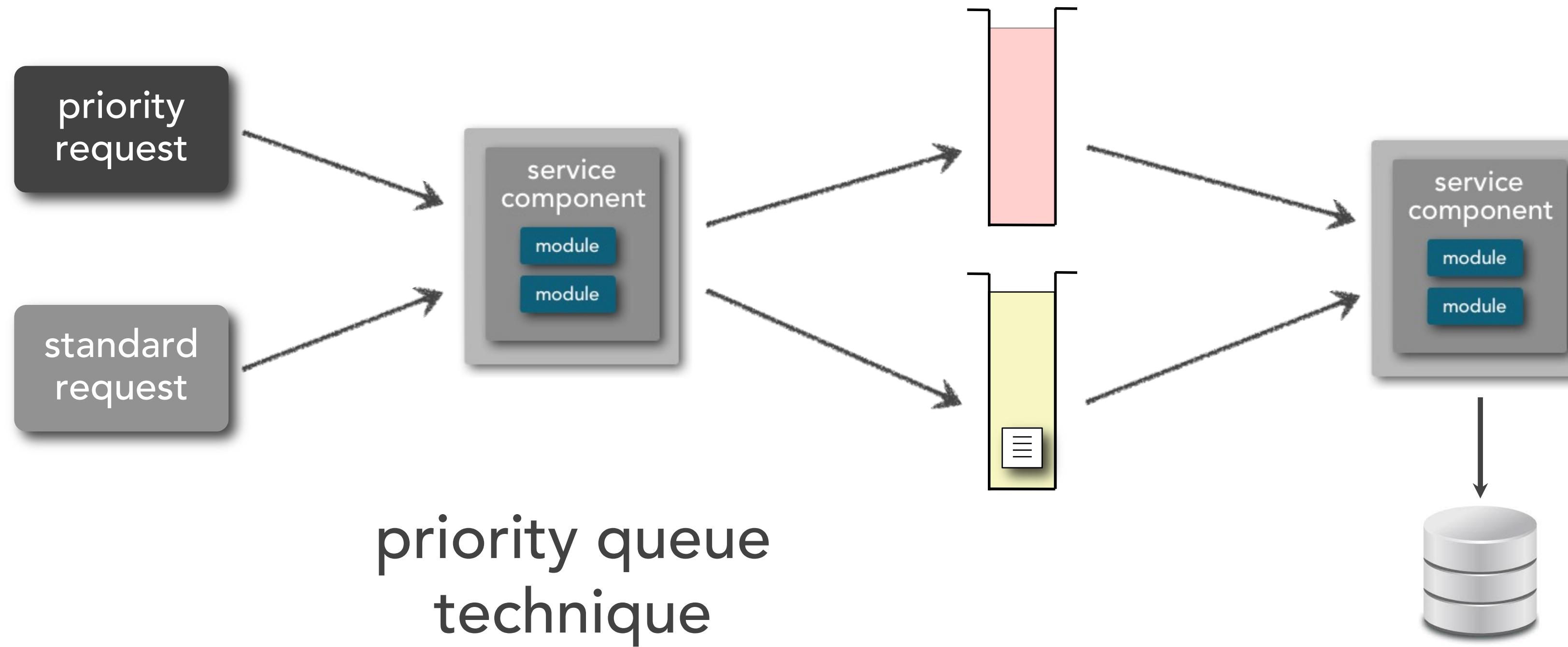
# ambulance pattern



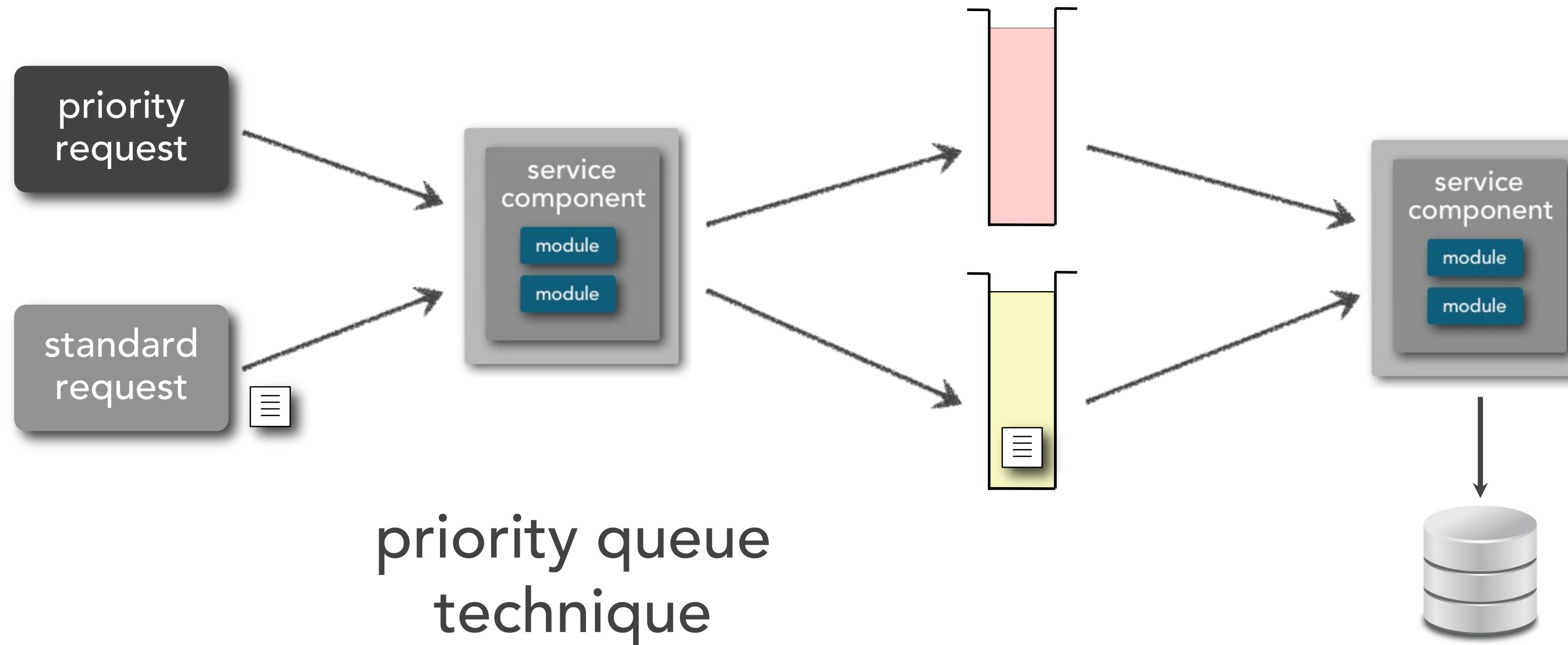
# ambulance pattern



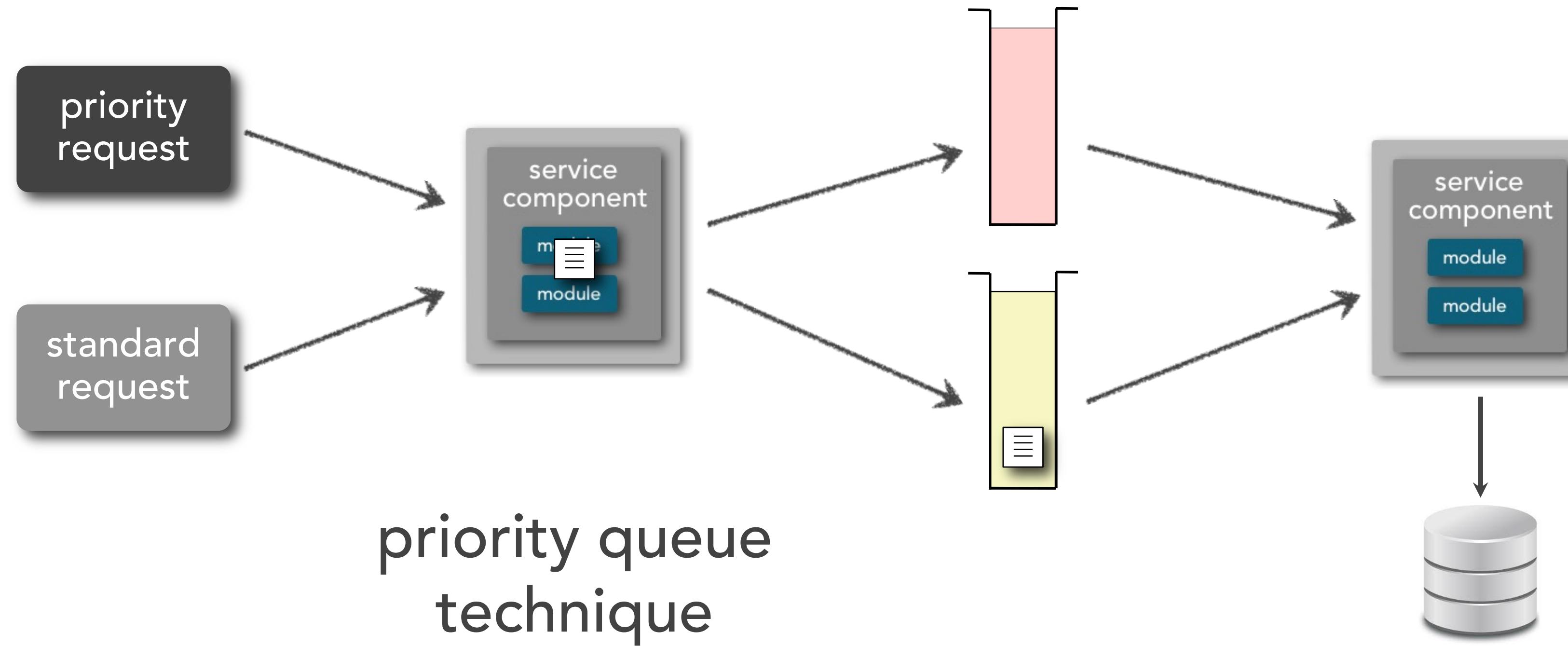
# ambulance pattern



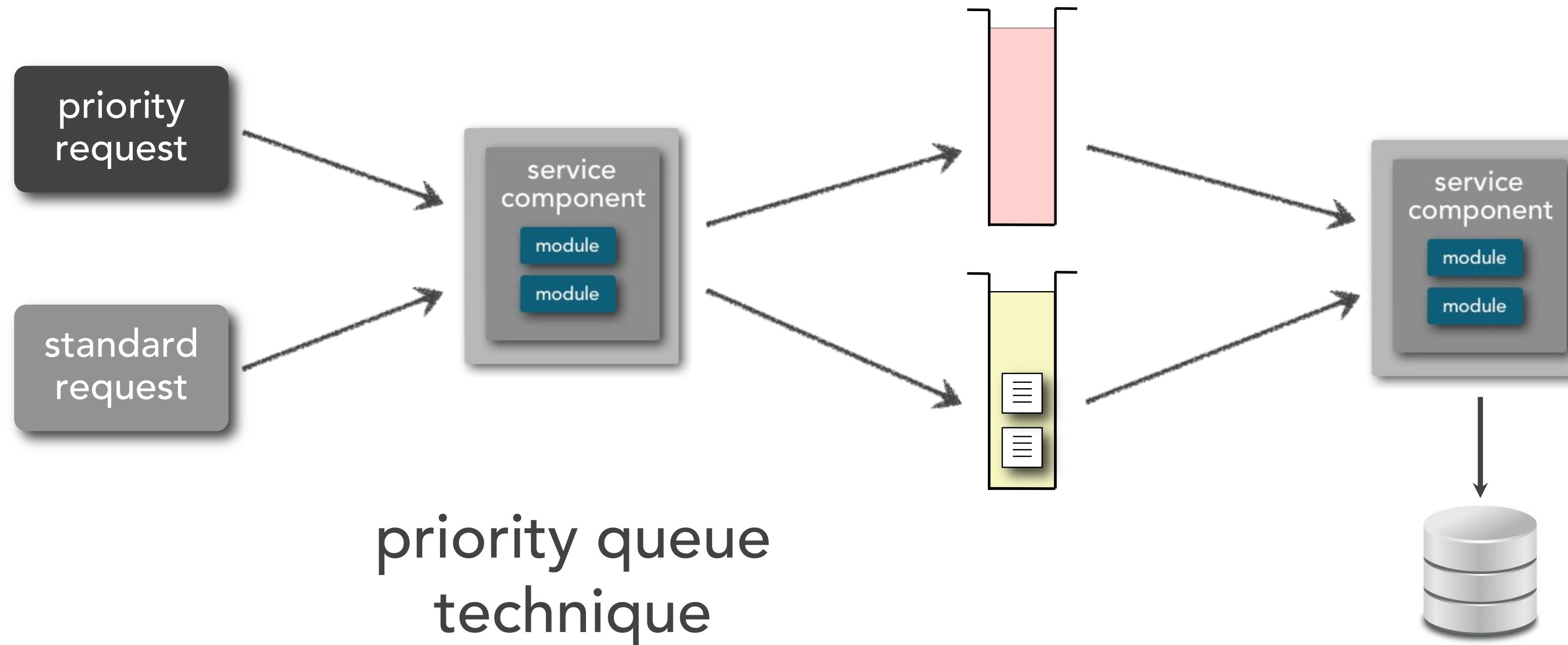
# ambulance pattern



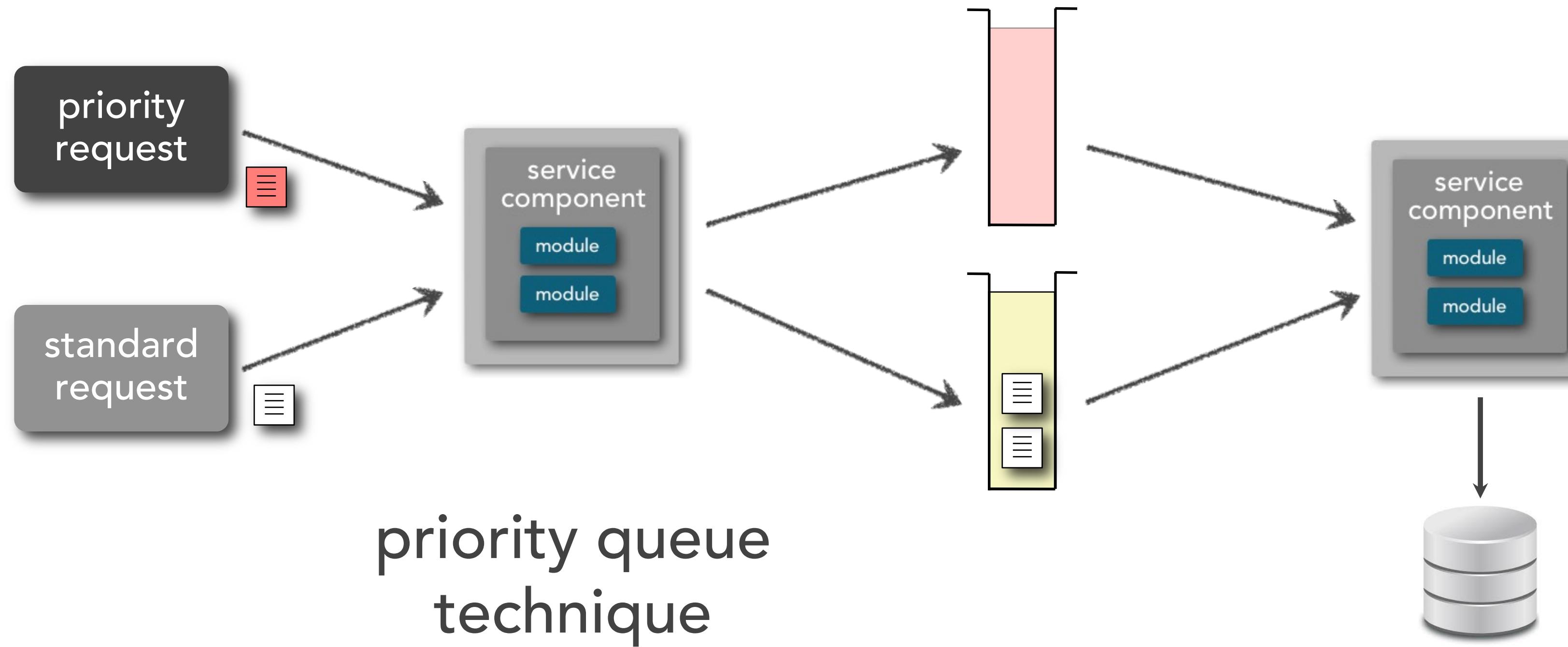
# ambulance pattern



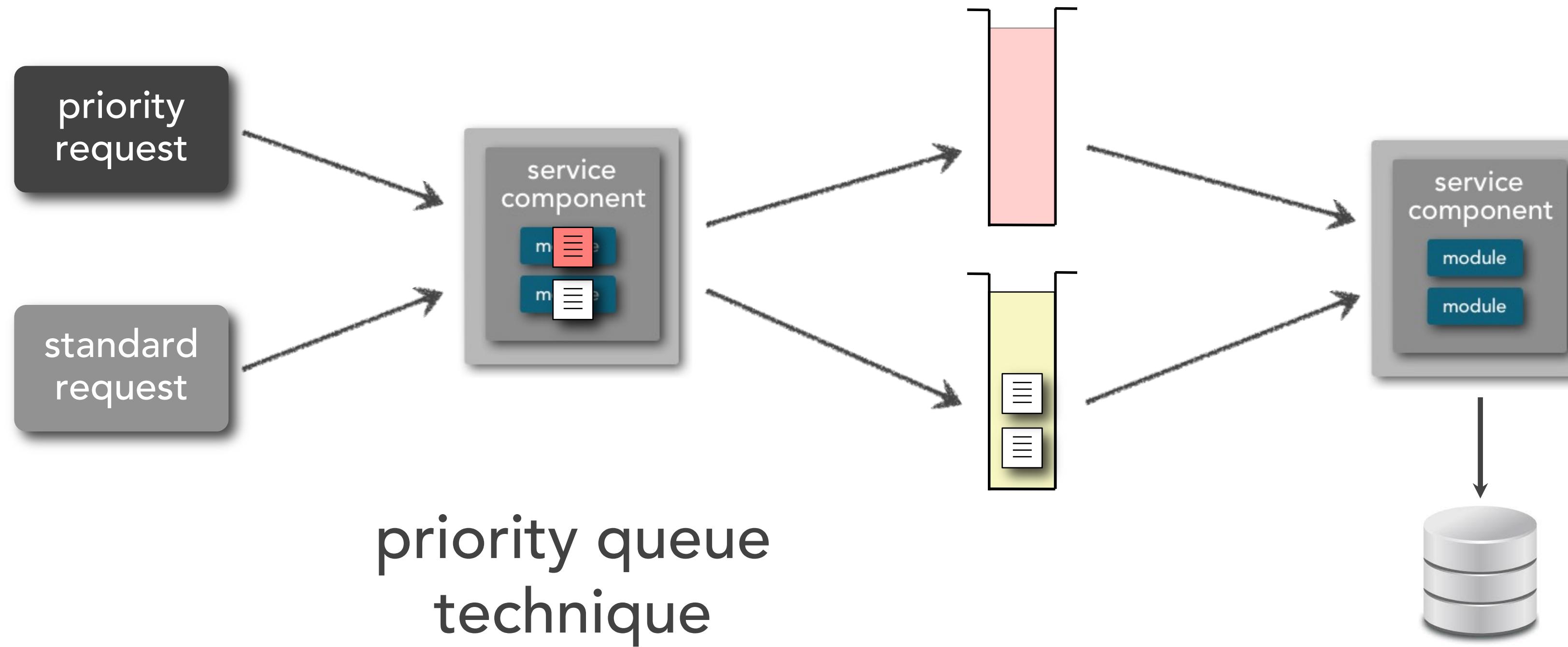
# ambulance pattern



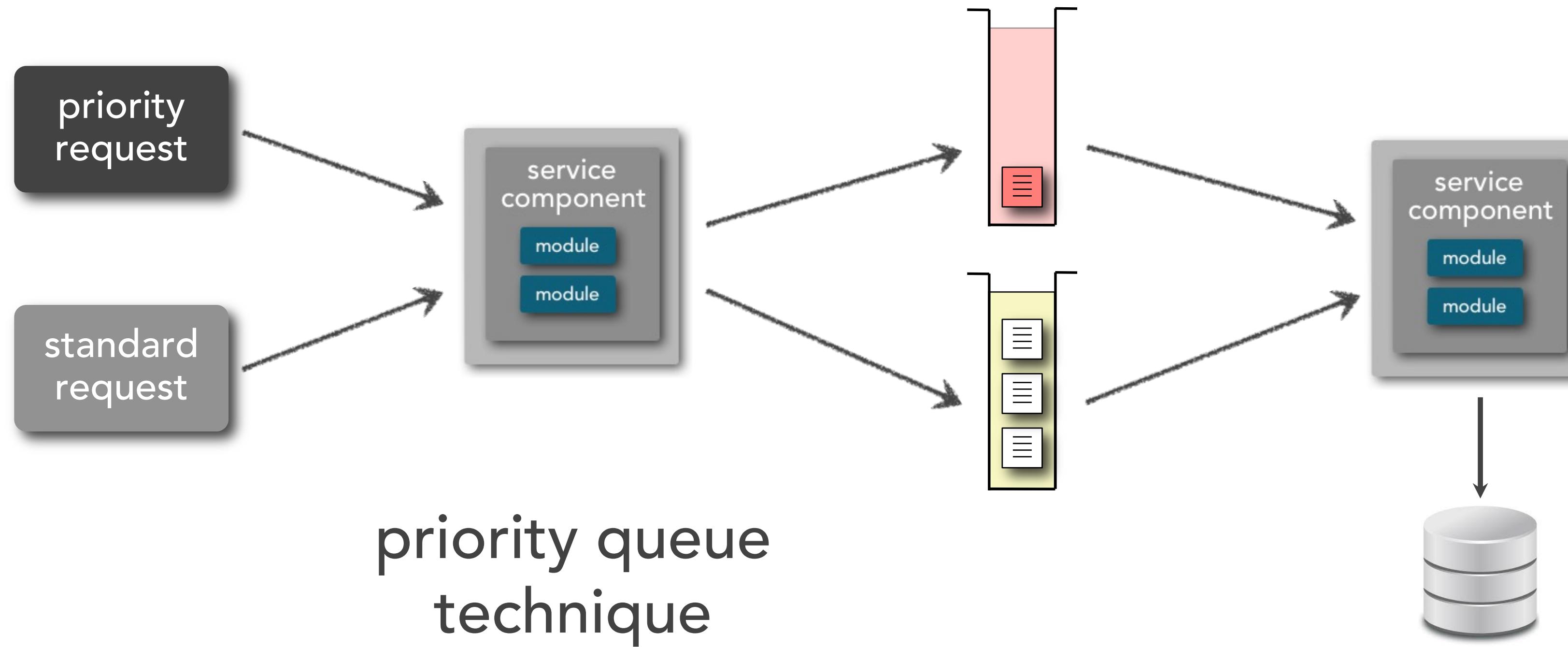
# ambulance pattern



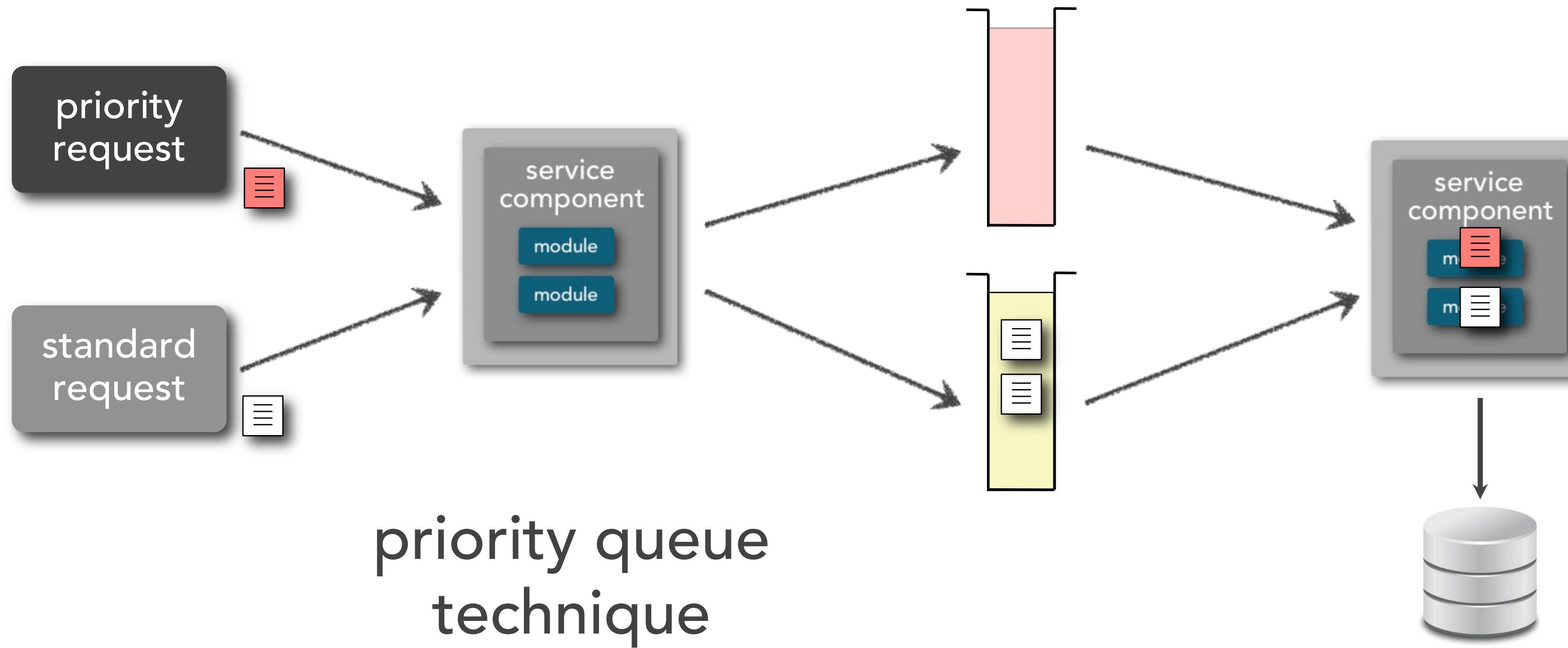
# ambulance pattern



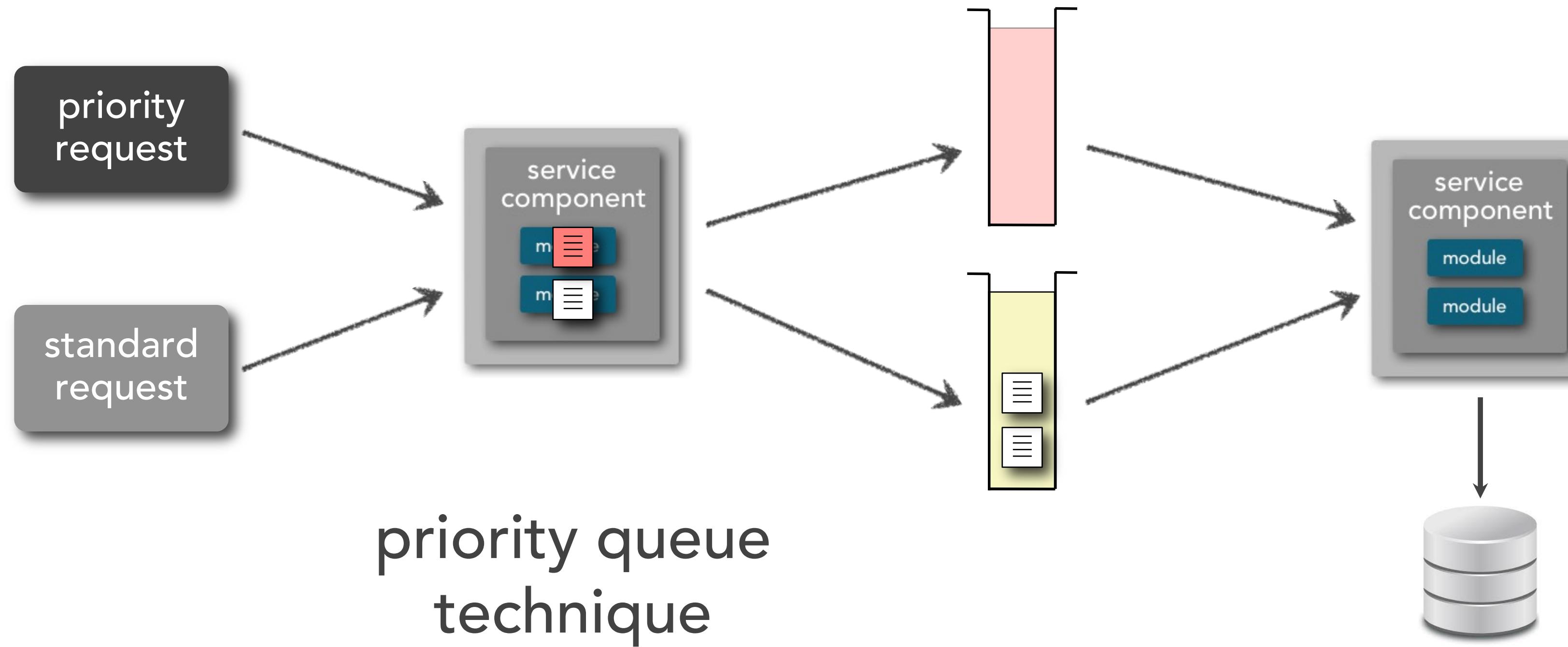
# ambulance pattern



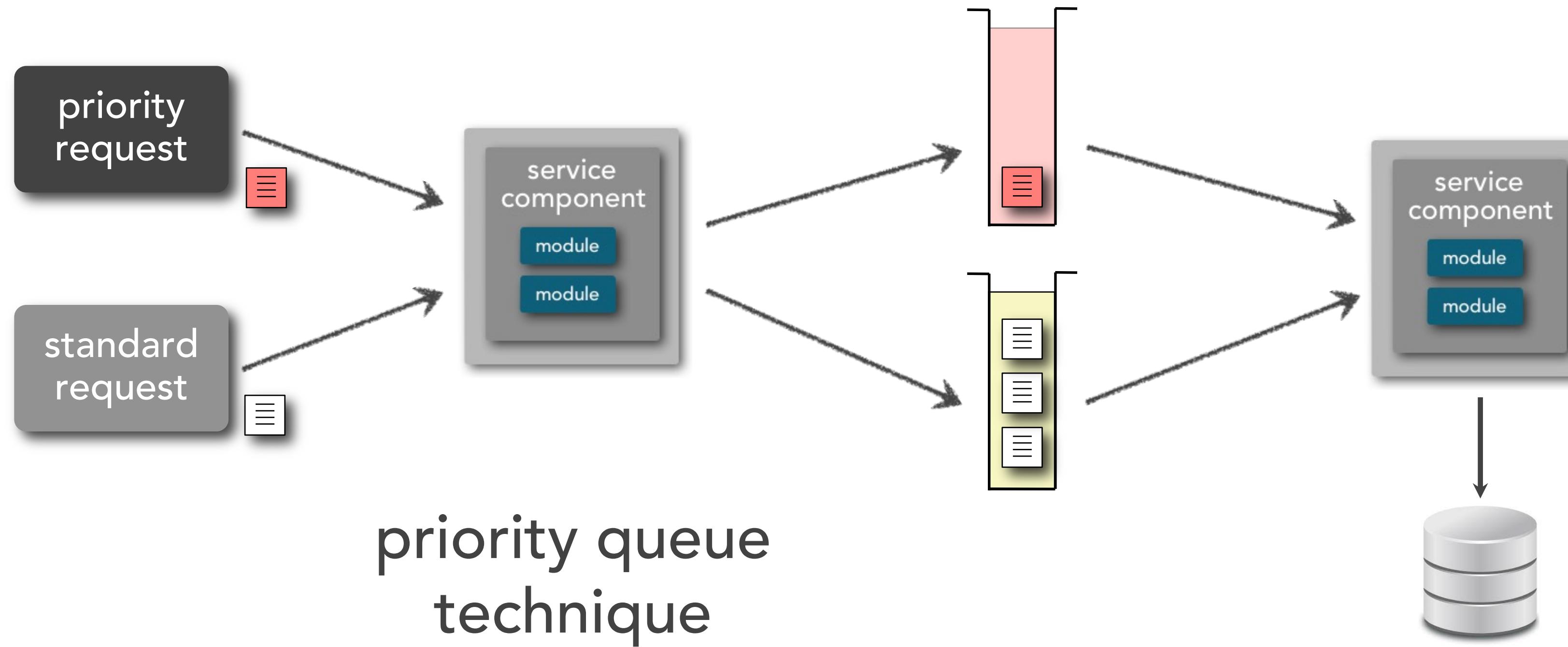
# ambulance pattern



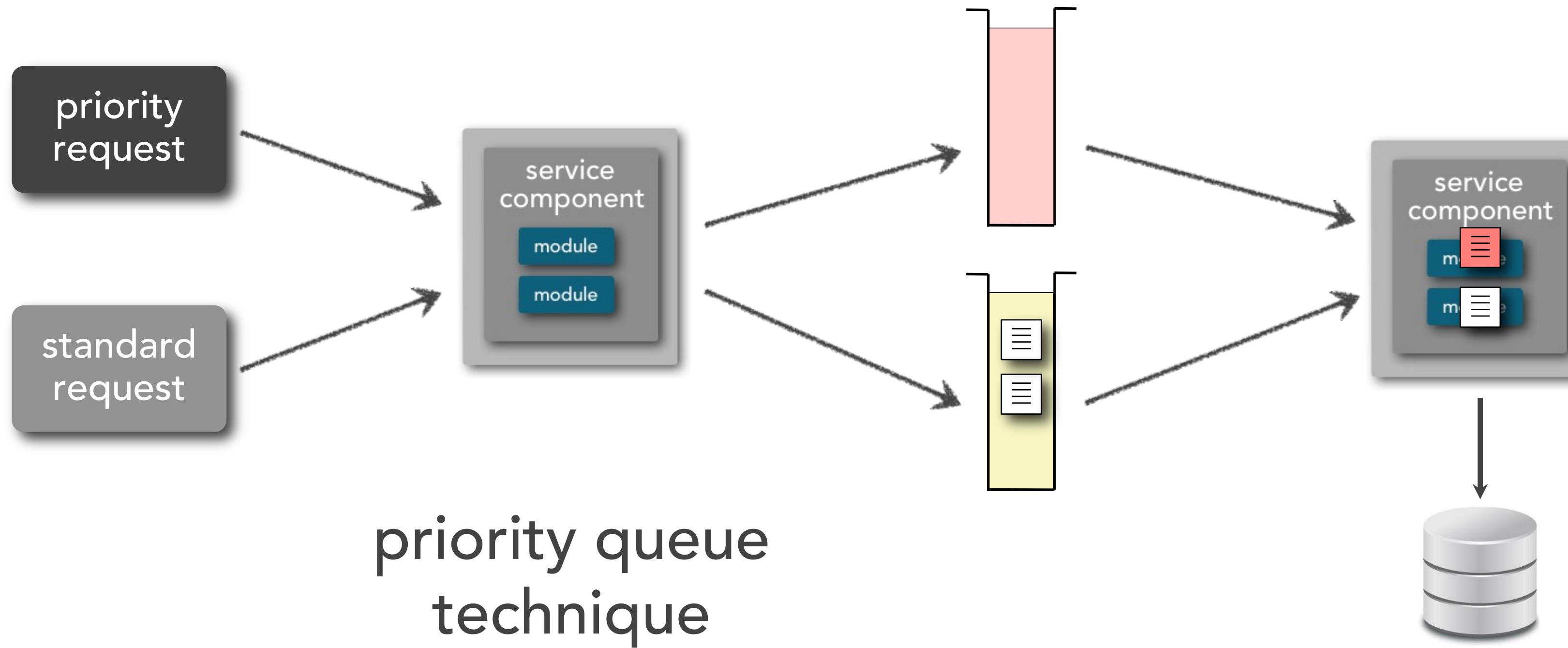
# ambulance pattern



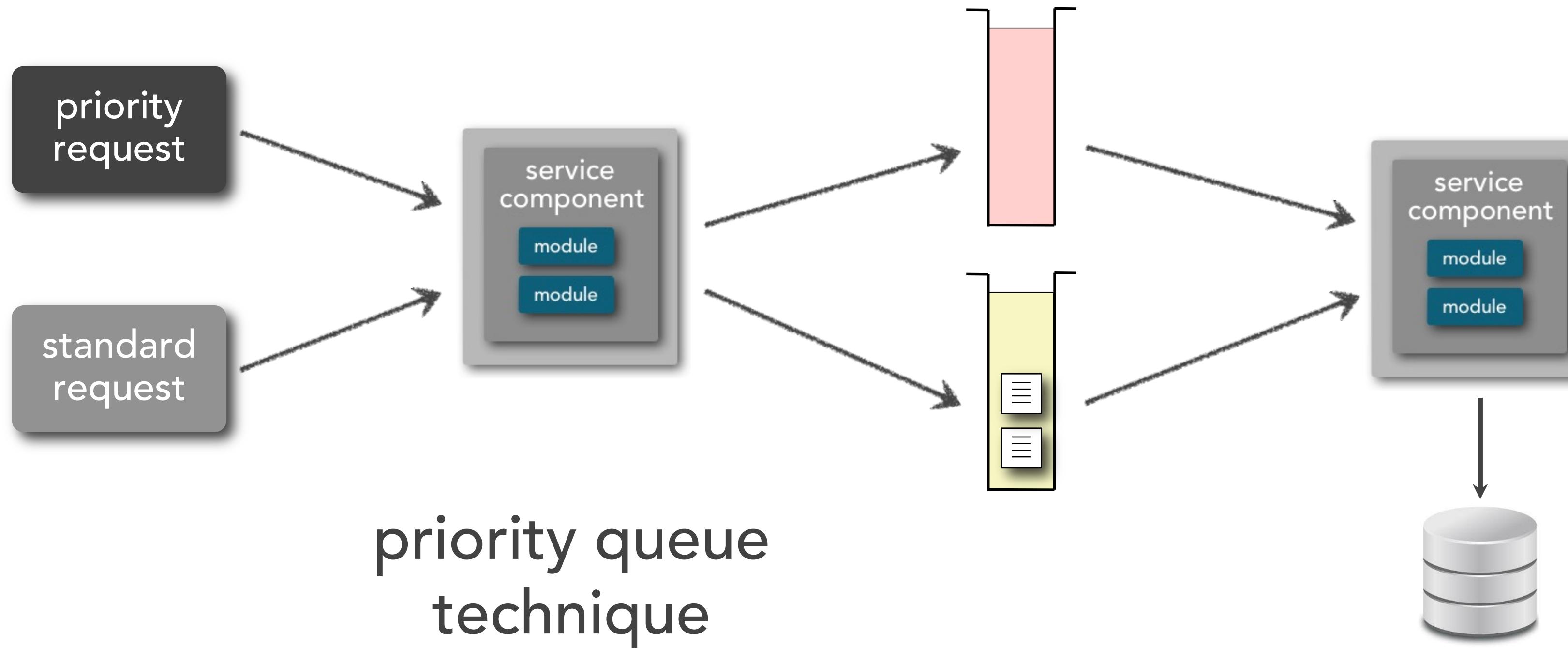
# ambulance pattern



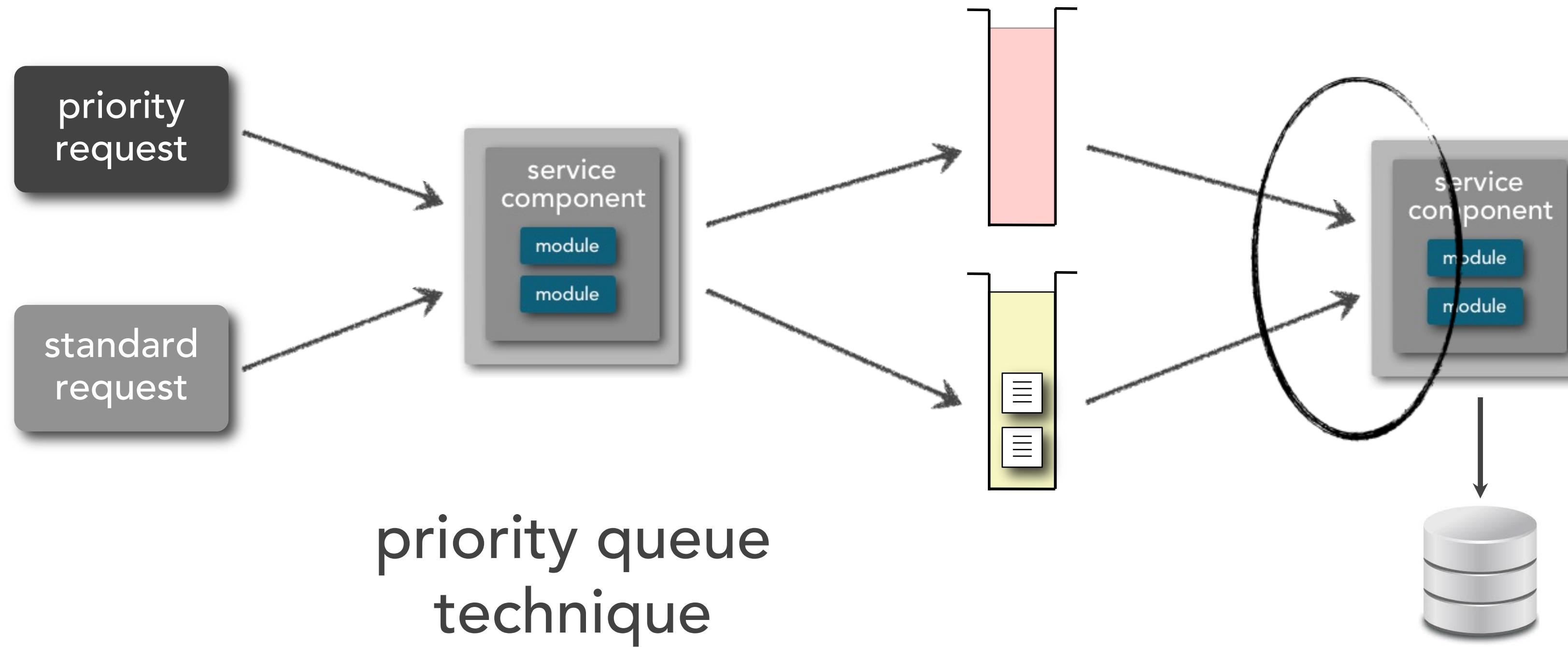
# ambulance pattern



# ambulance pattern

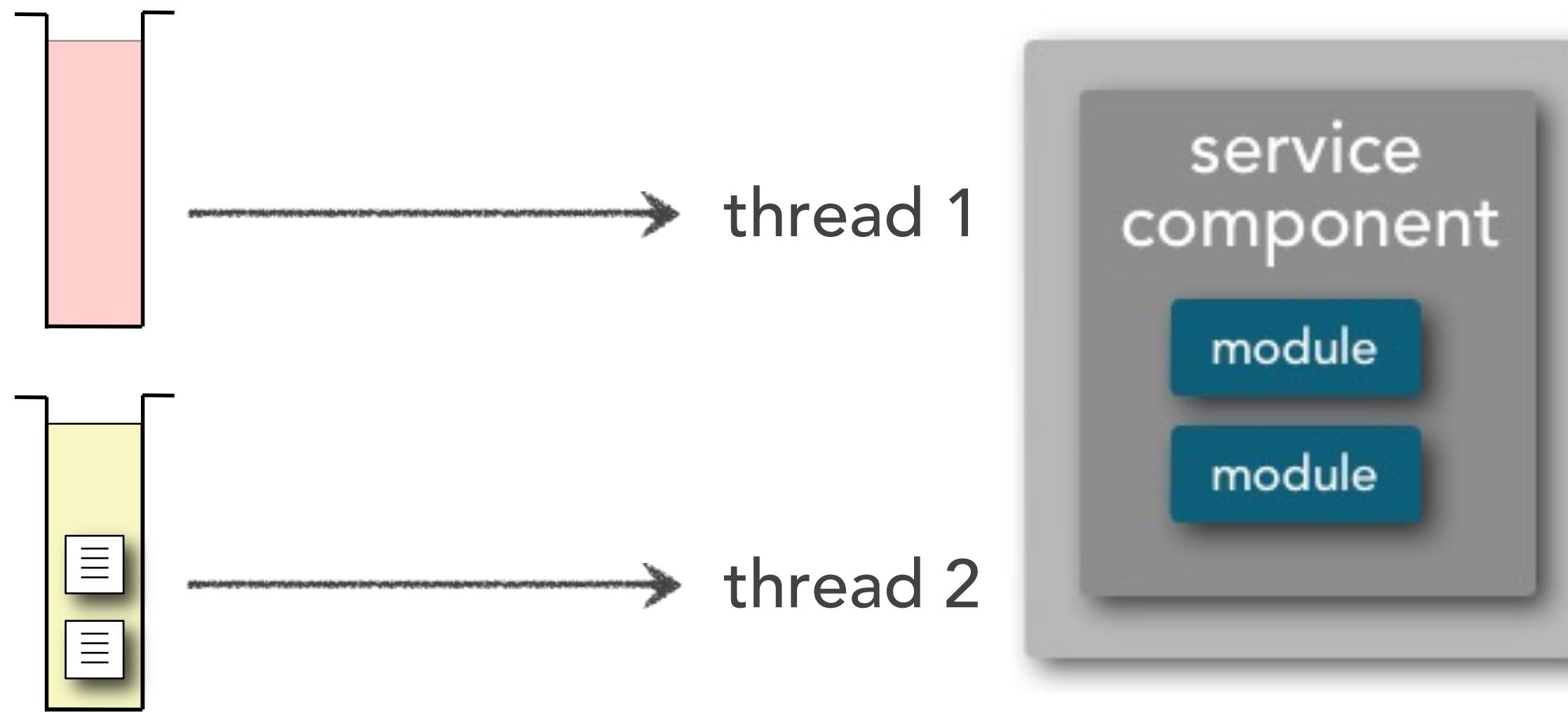


# ambulance pattern



# ambulance pattern

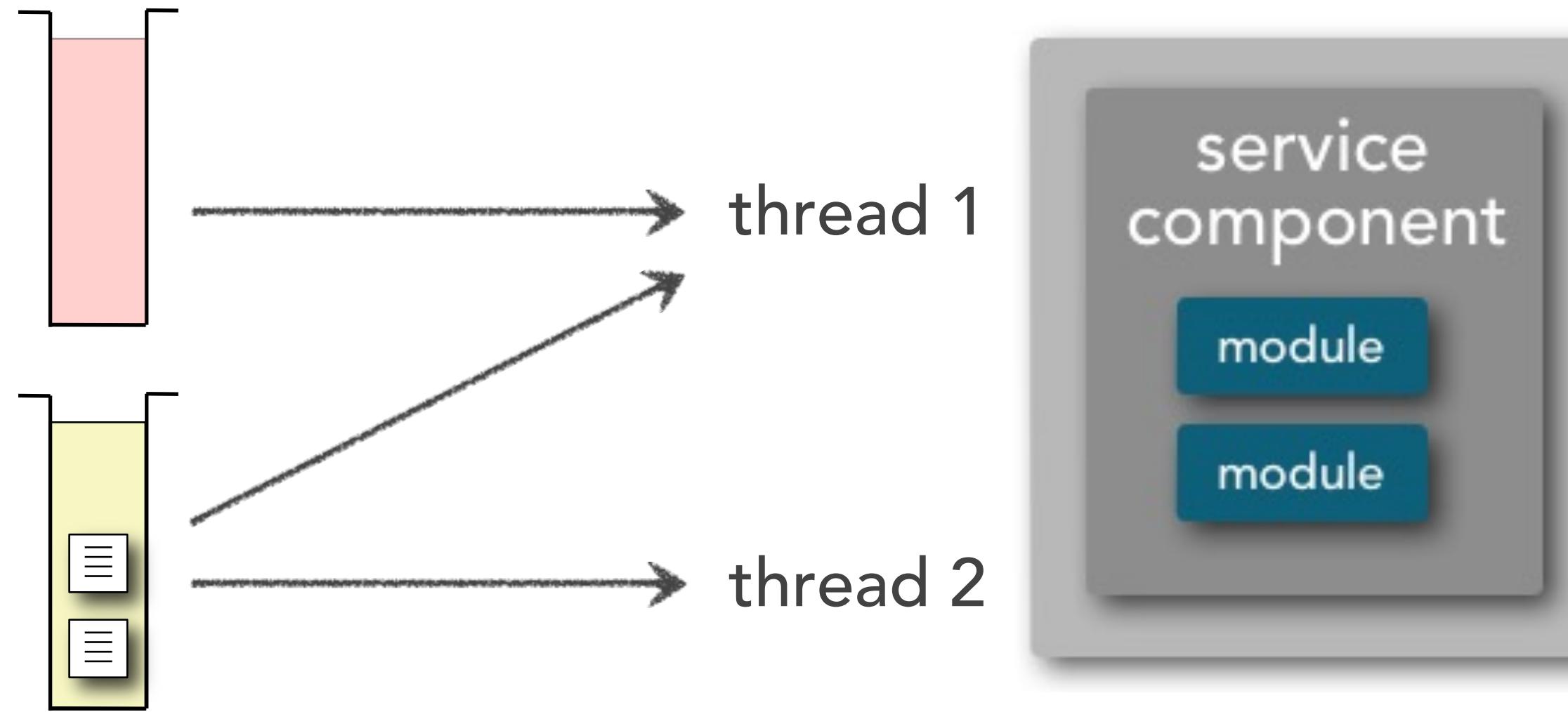
## carpool vs. ambulance



**carpool** - thread 1 is dedicated to priority messages only and remains idle if no priority messages exist

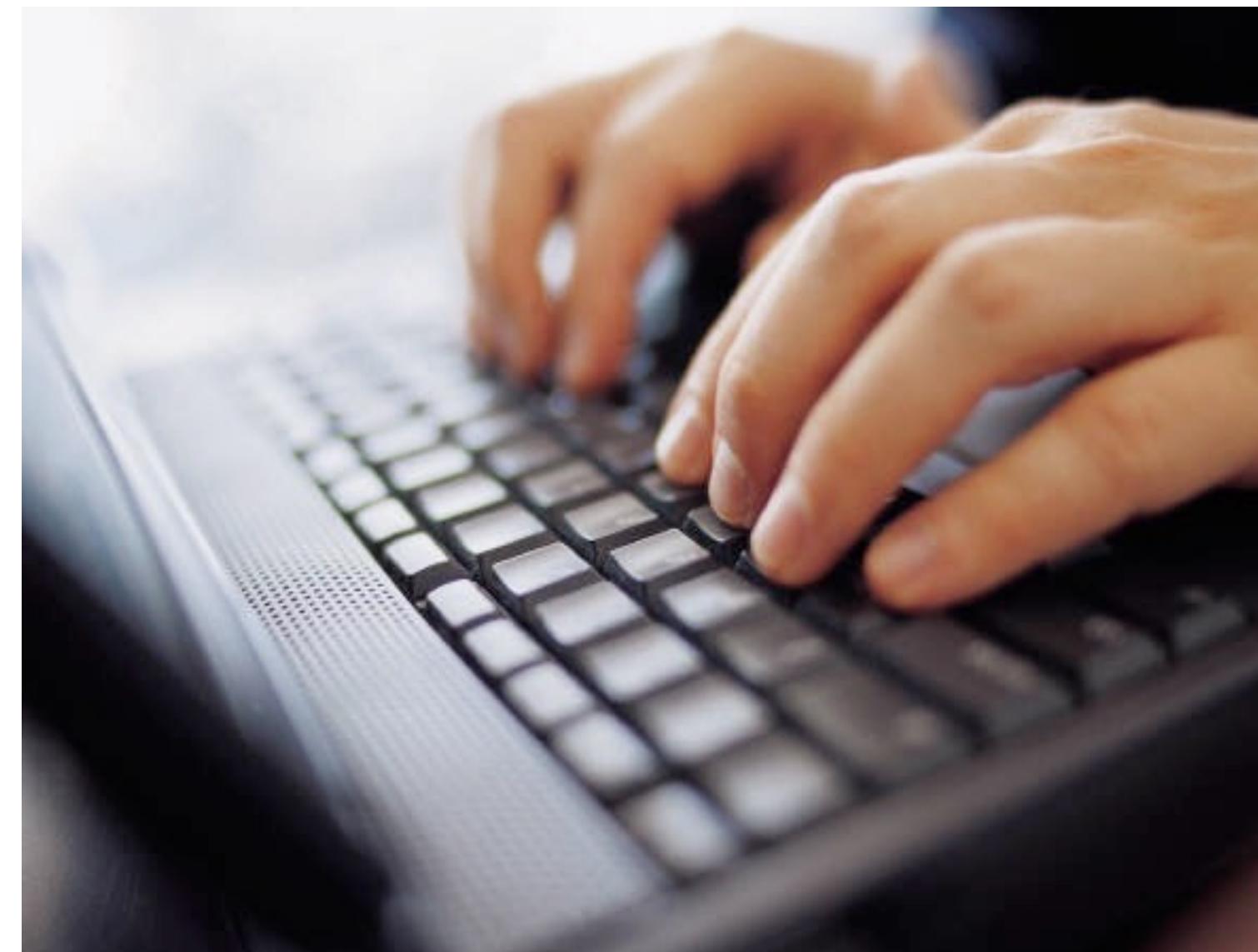
# ambulance pattern

## carpool vs. ambulance



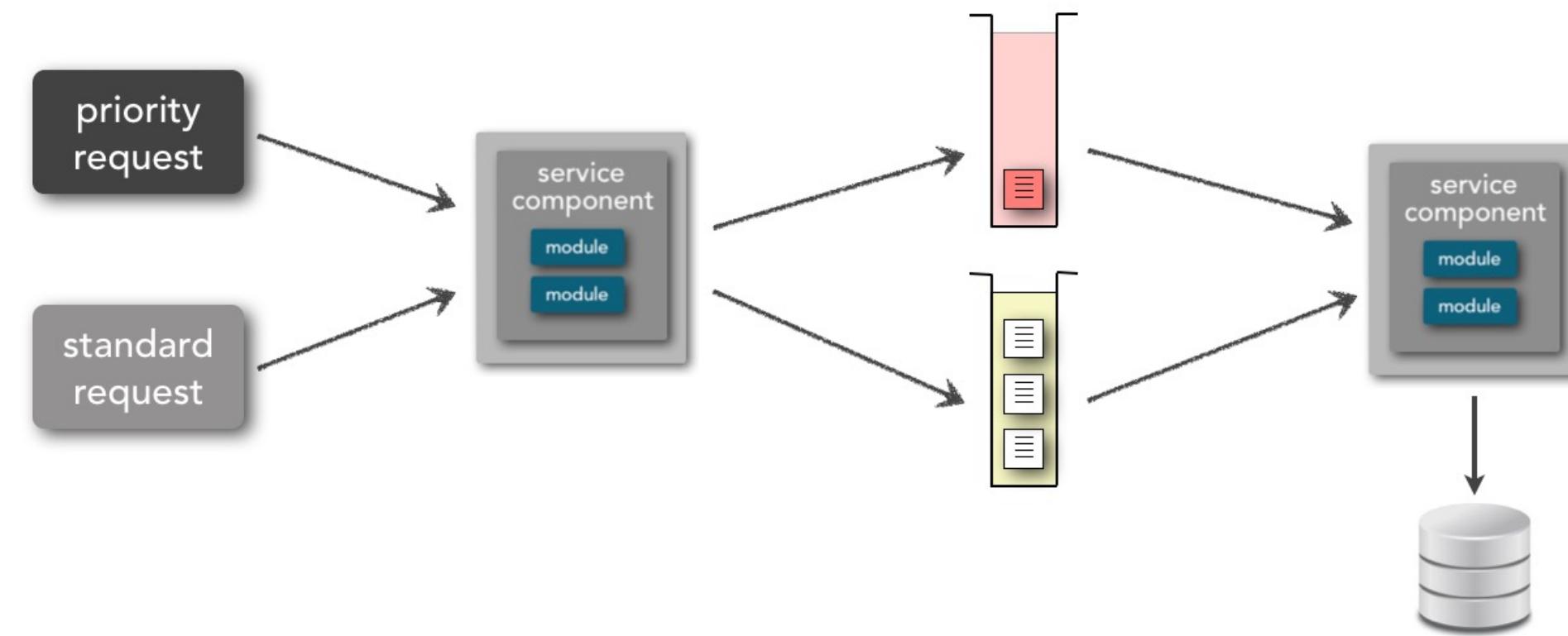
**ambulance** - thread 1 can handle standard requests as well, but always checks priority queue first before getting the next standard message

# ambulance pattern

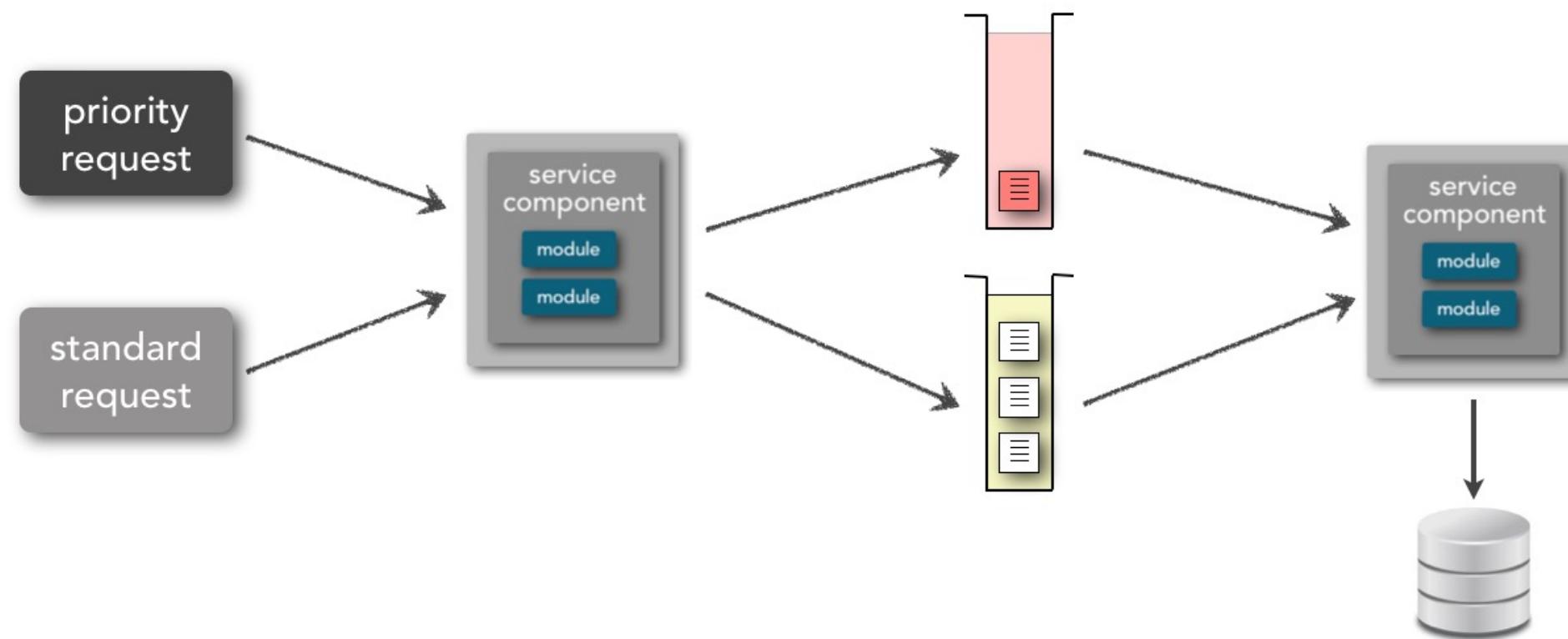


let's apply the pattern...

# ambulance pattern



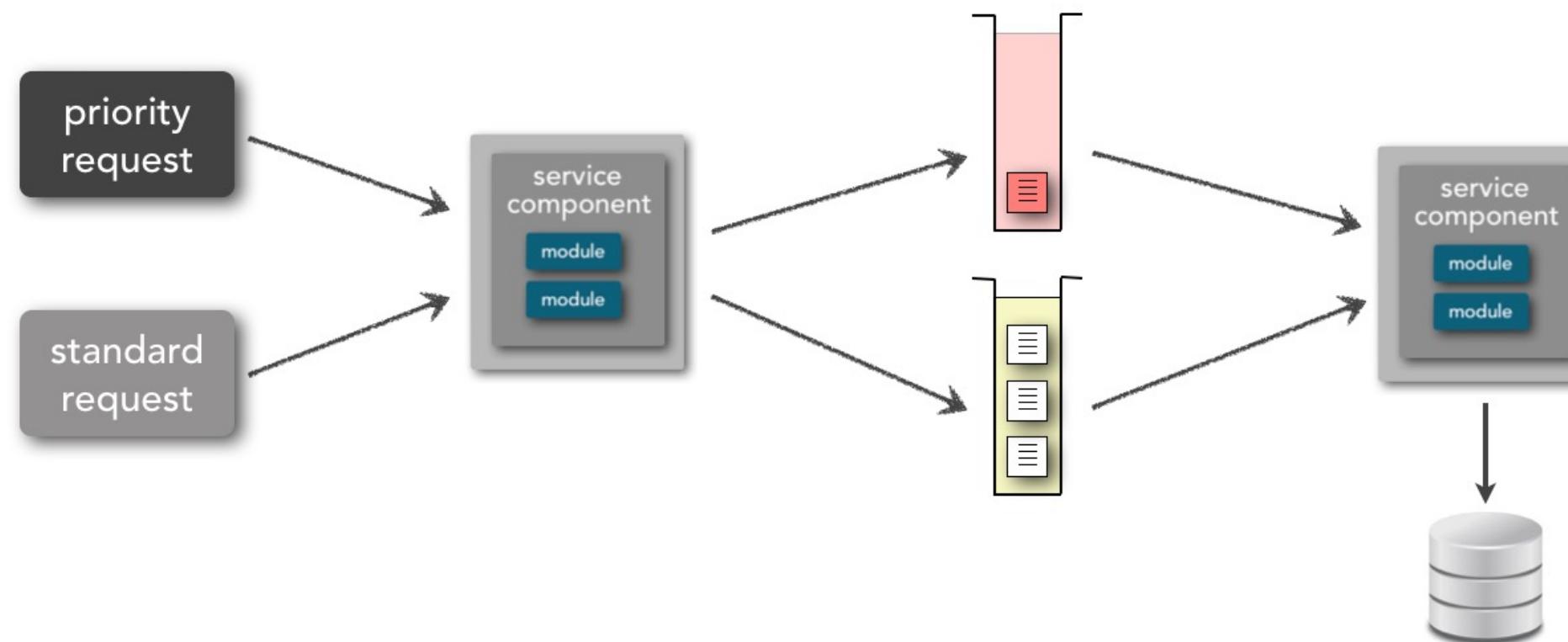
# ambulance pattern



message priority  
throughput  
performance



# ambulance pattern



message priority  
throughput  
performance

complexity



# Watch Notification Pattern

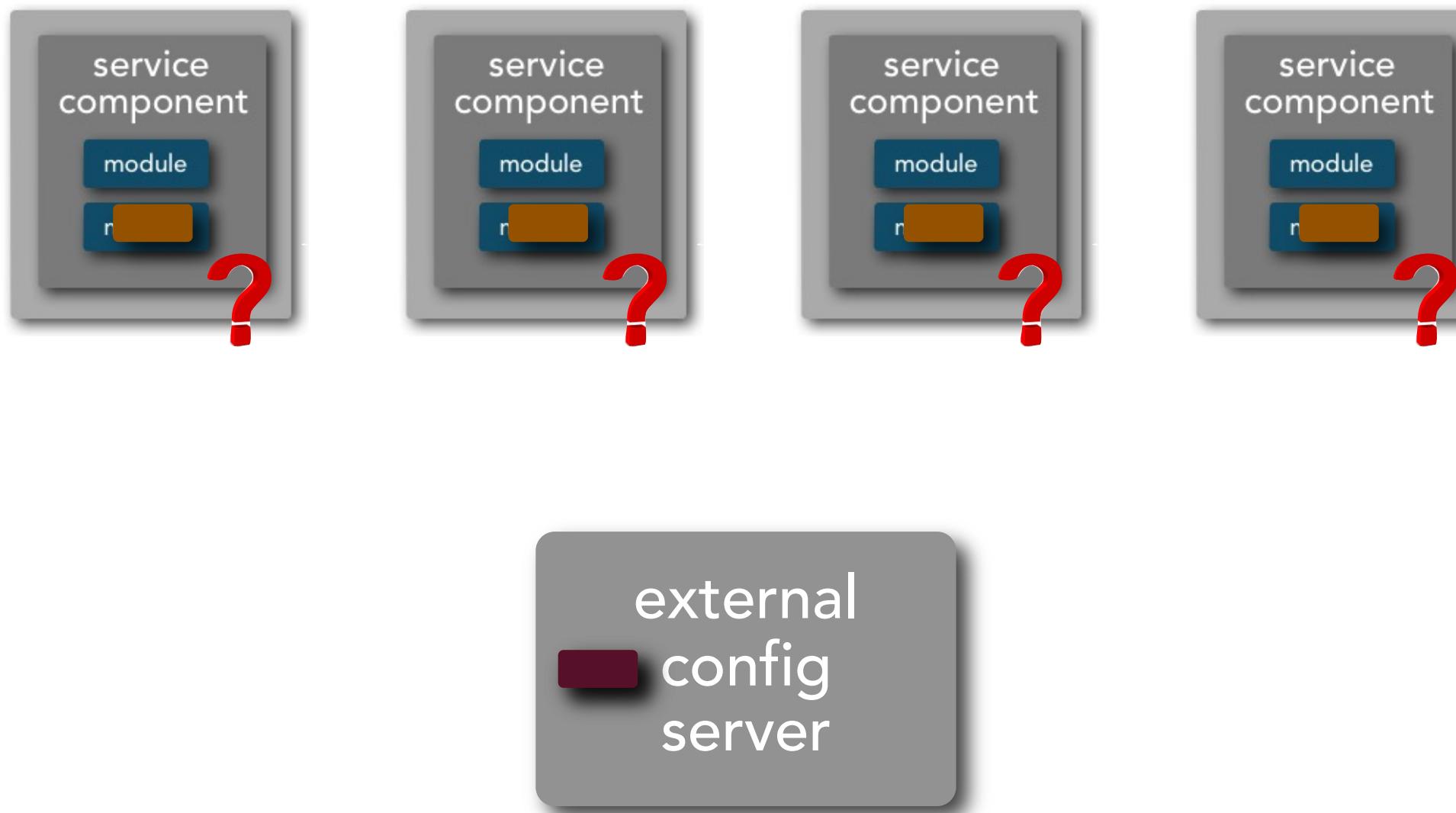
# watch notification pattern



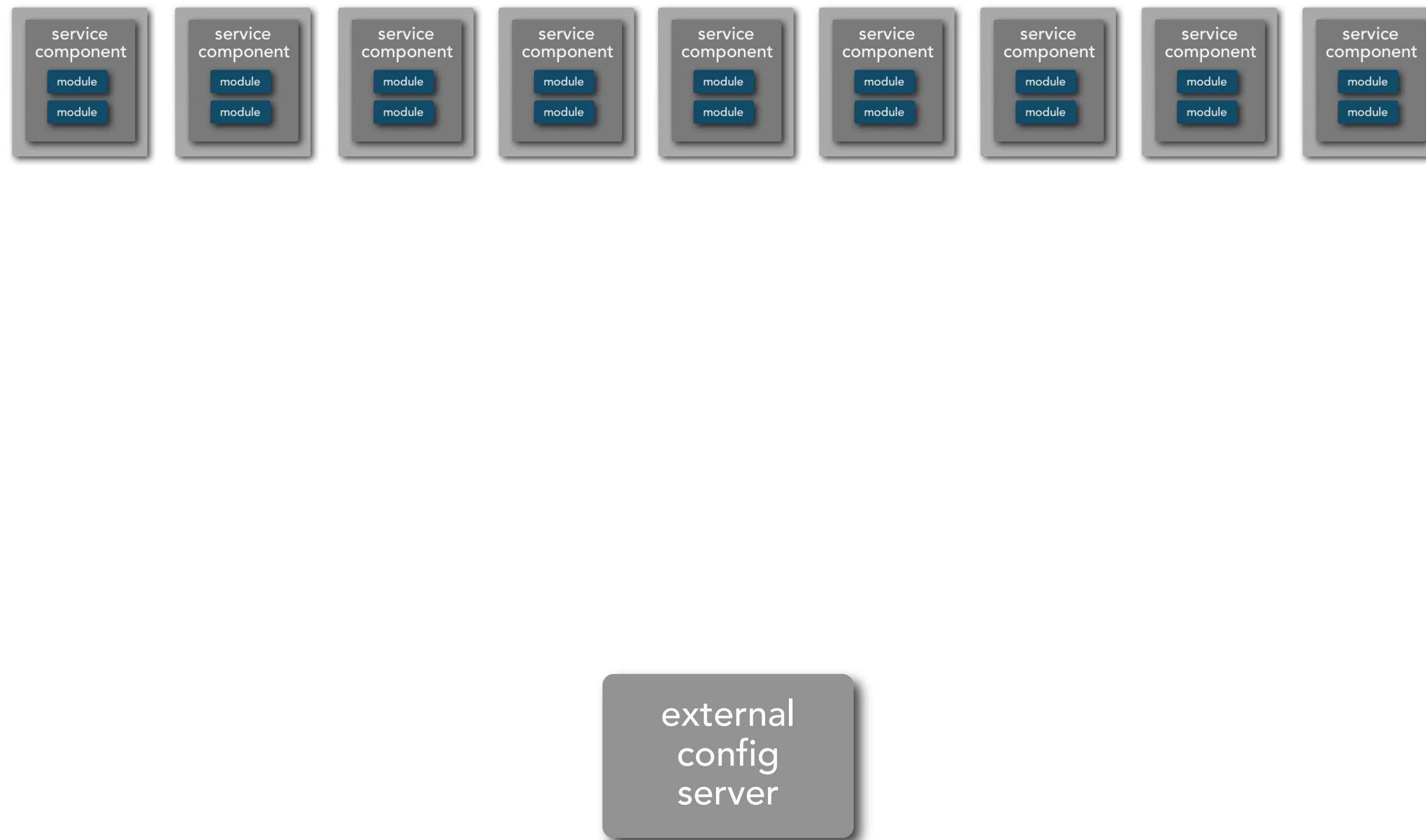
<https://github.com/wmr513/event-driven-patterns/tree/master/watch>

# watch notification pattern

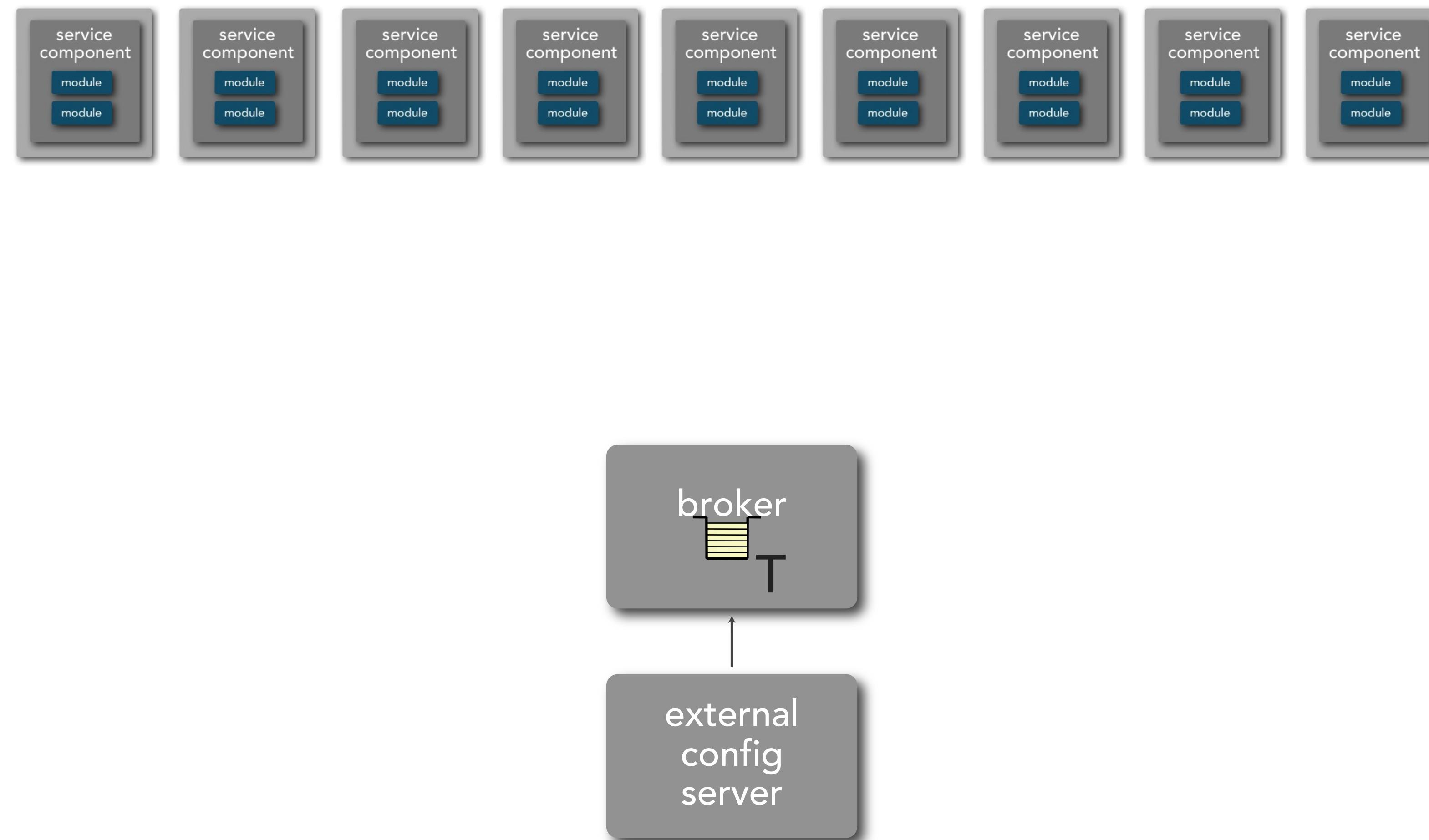
*“how can I send notification events to services without maintaining a persistent connection to those services?”*



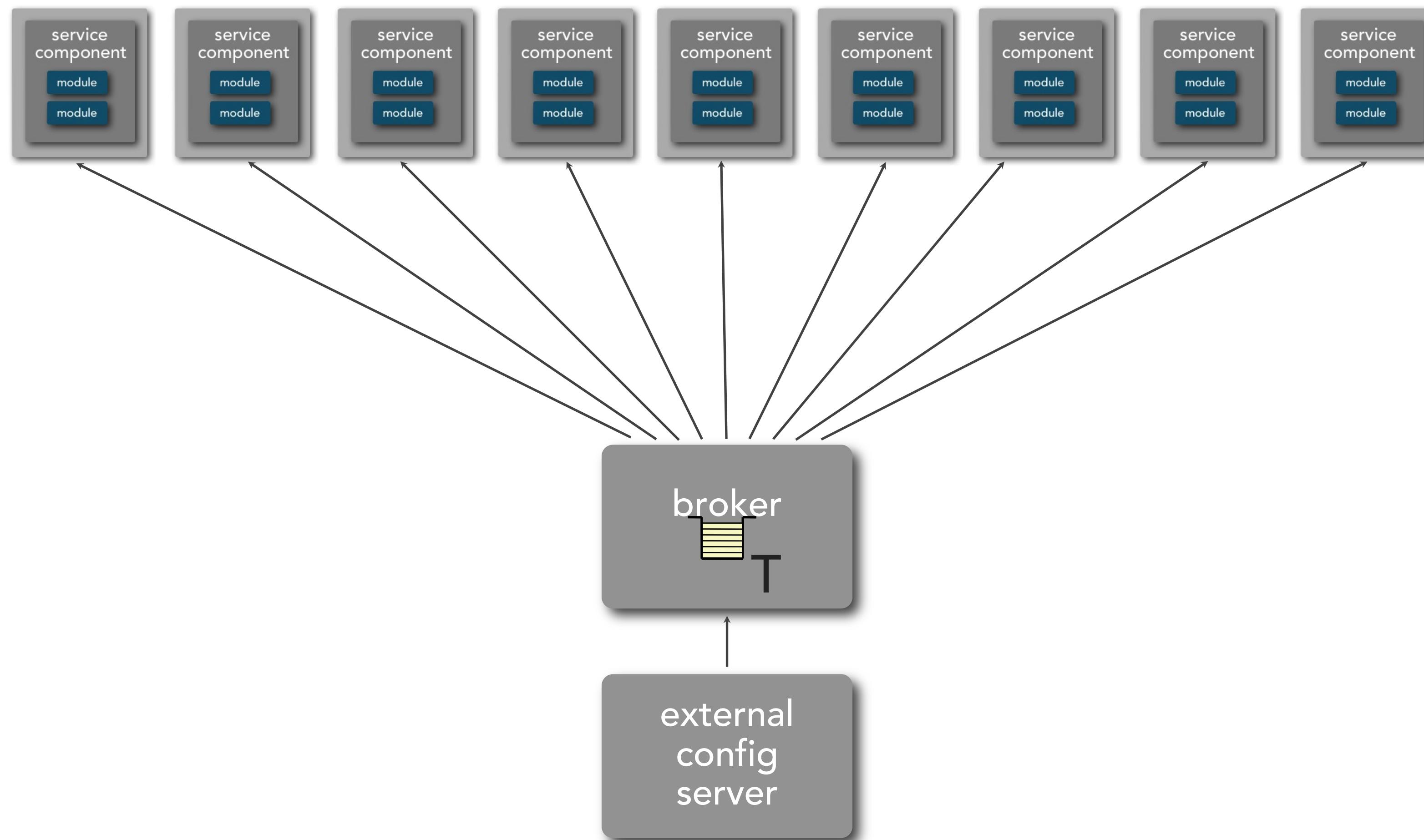
# watch notification pattern



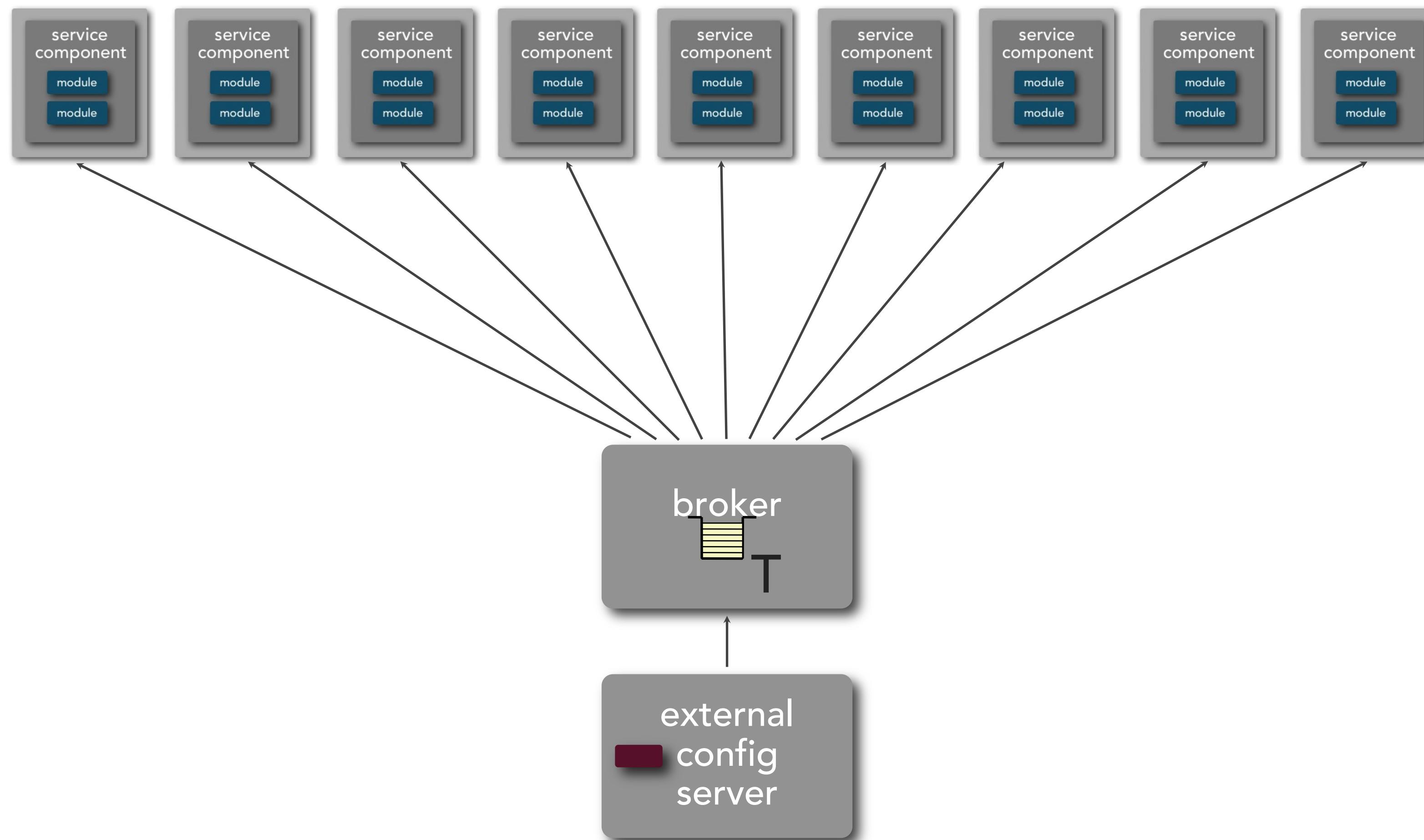
# watch notification pattern



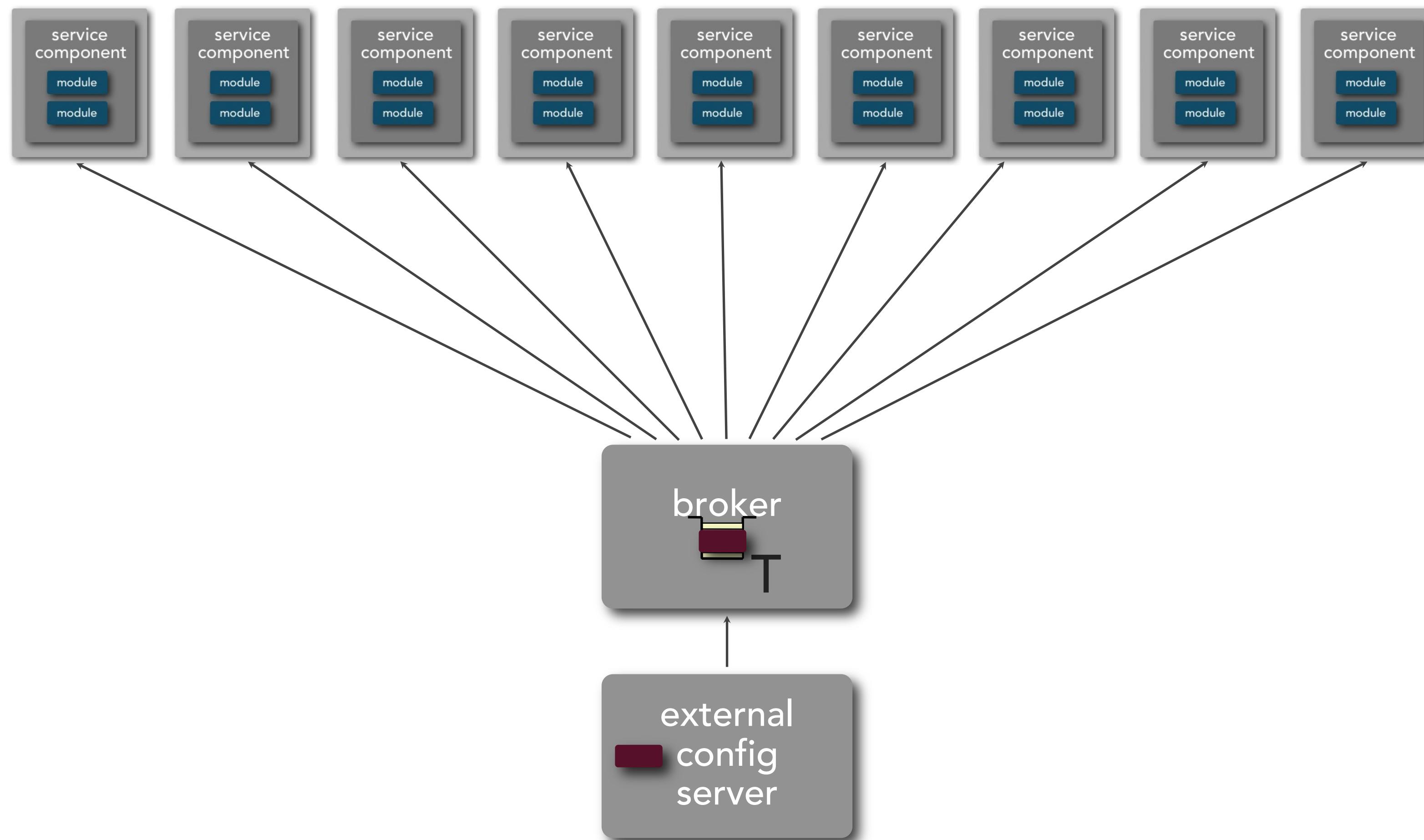
# watch notification pattern



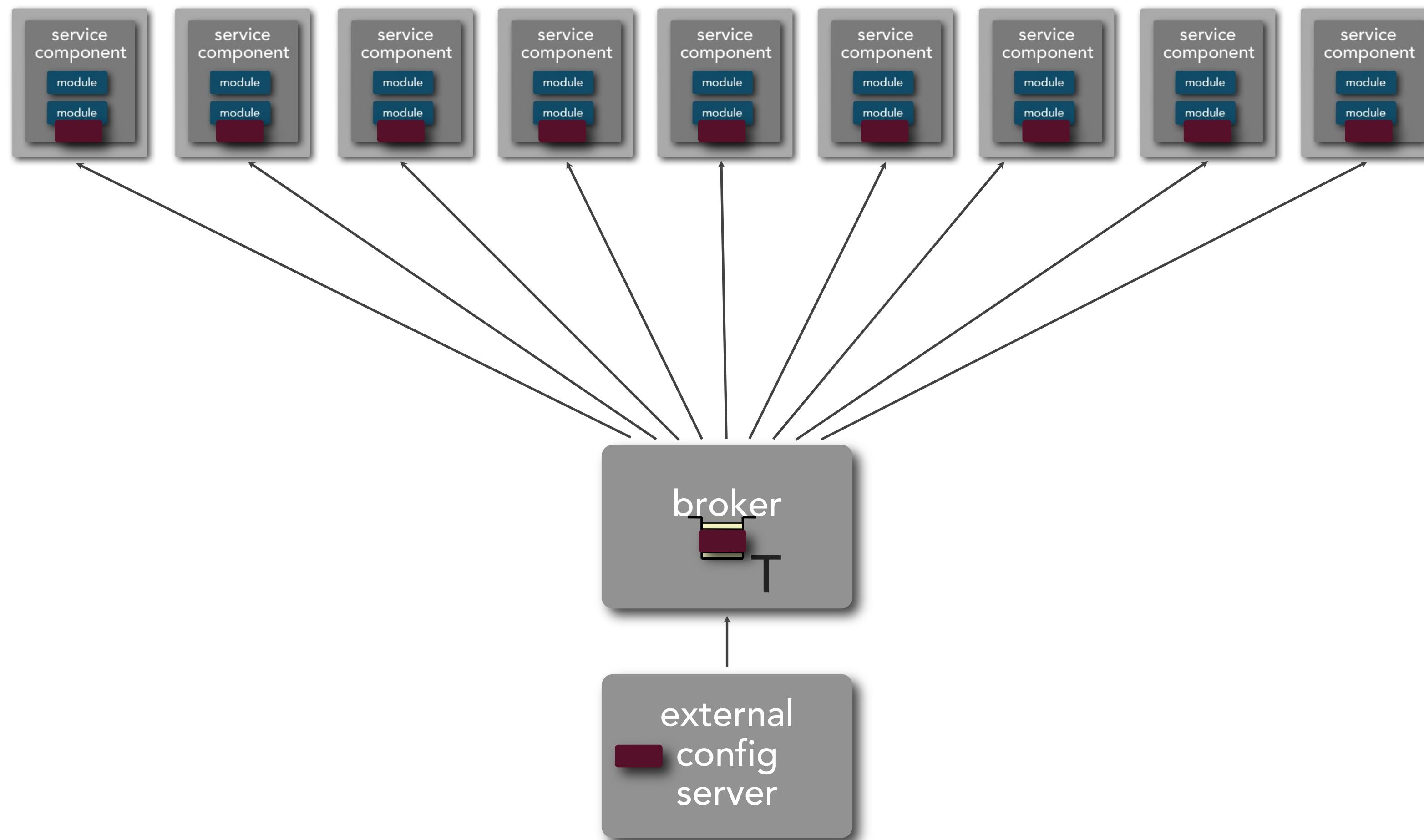
# watch notification pattern



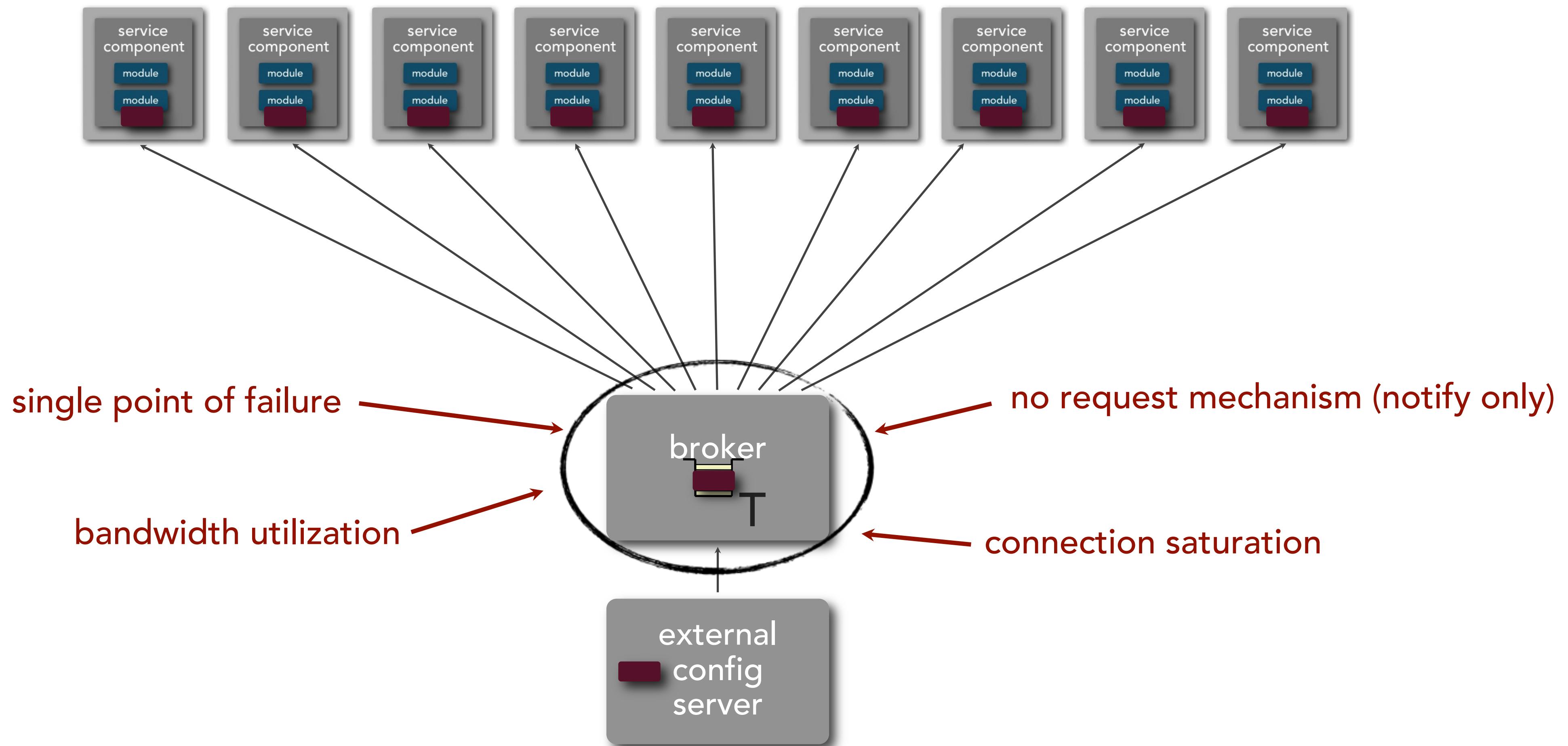
# watch notification pattern



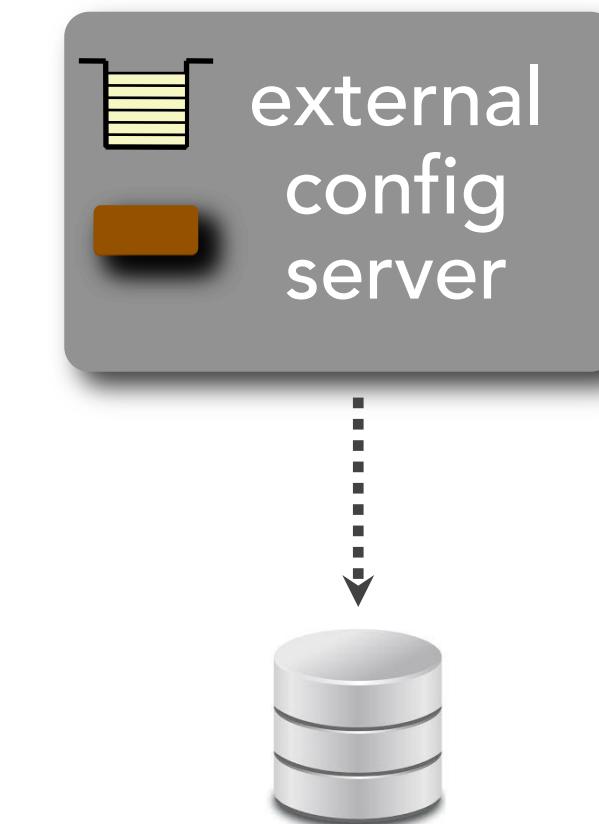
# watch notification pattern



# watch notification pattern

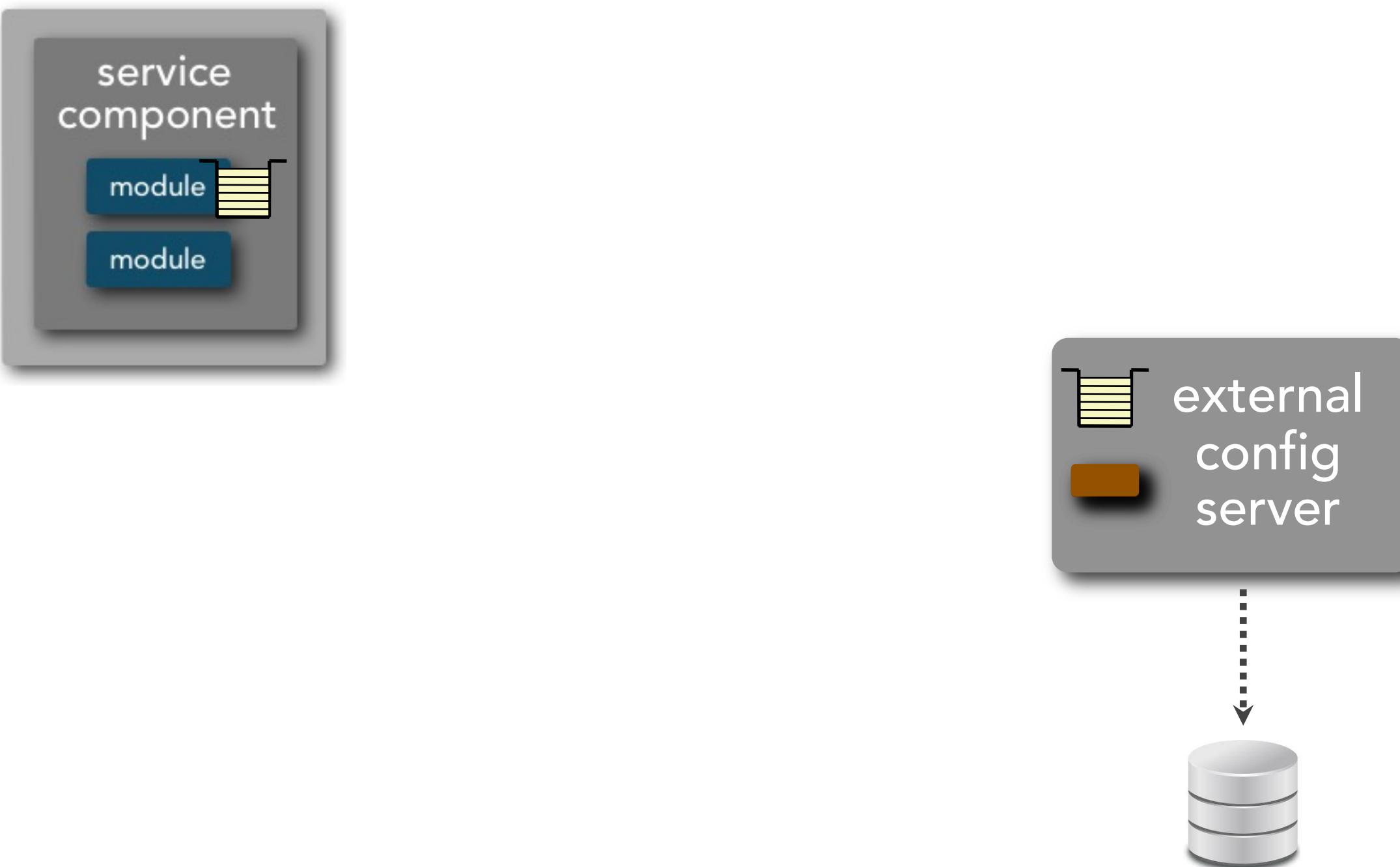


# watch notification pattern



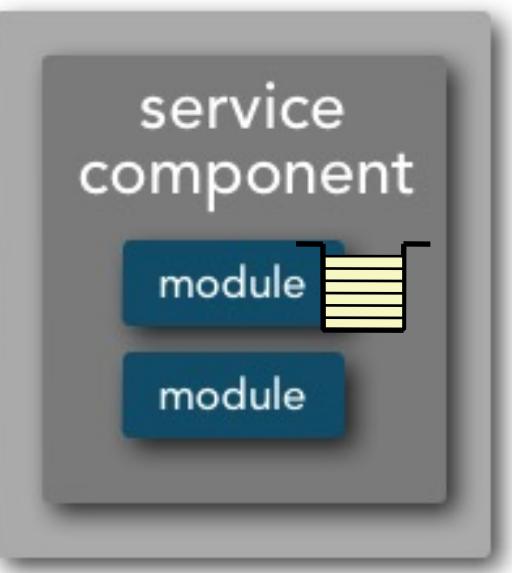
# watch notification pattern

svc1:62880/watch\_q

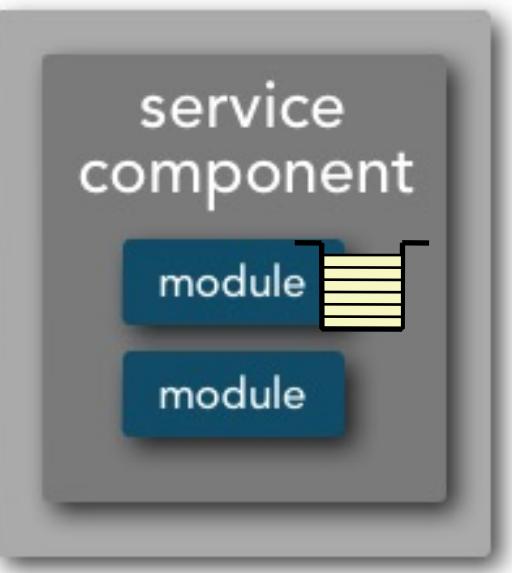


# watch notification pattern

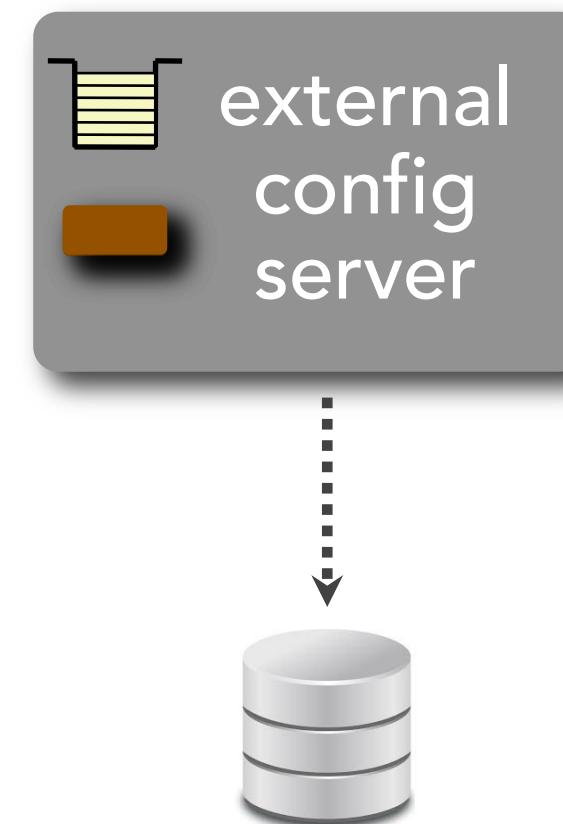
svc1:62880/watch\_q



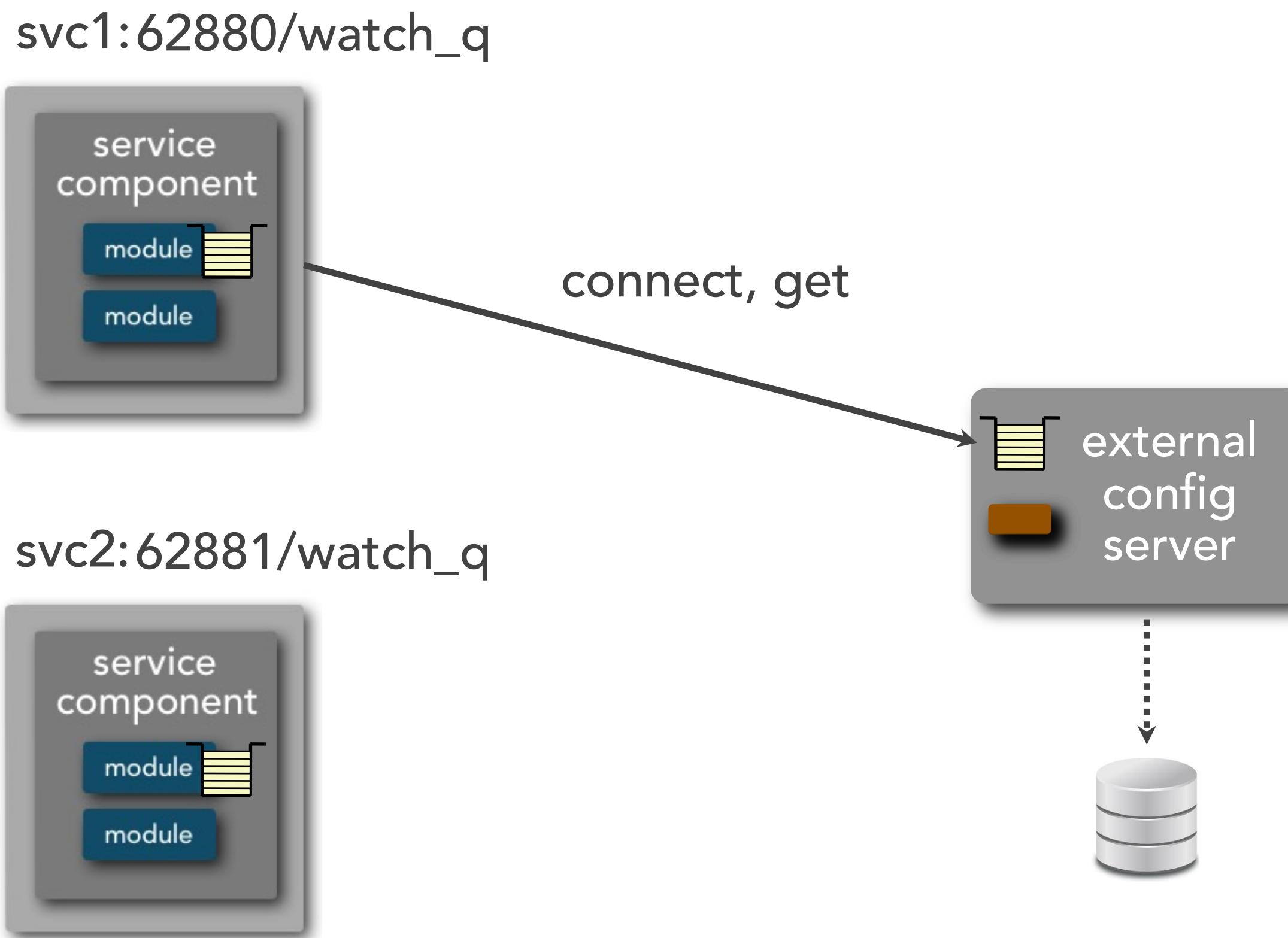
svc2:62881/watch\_q



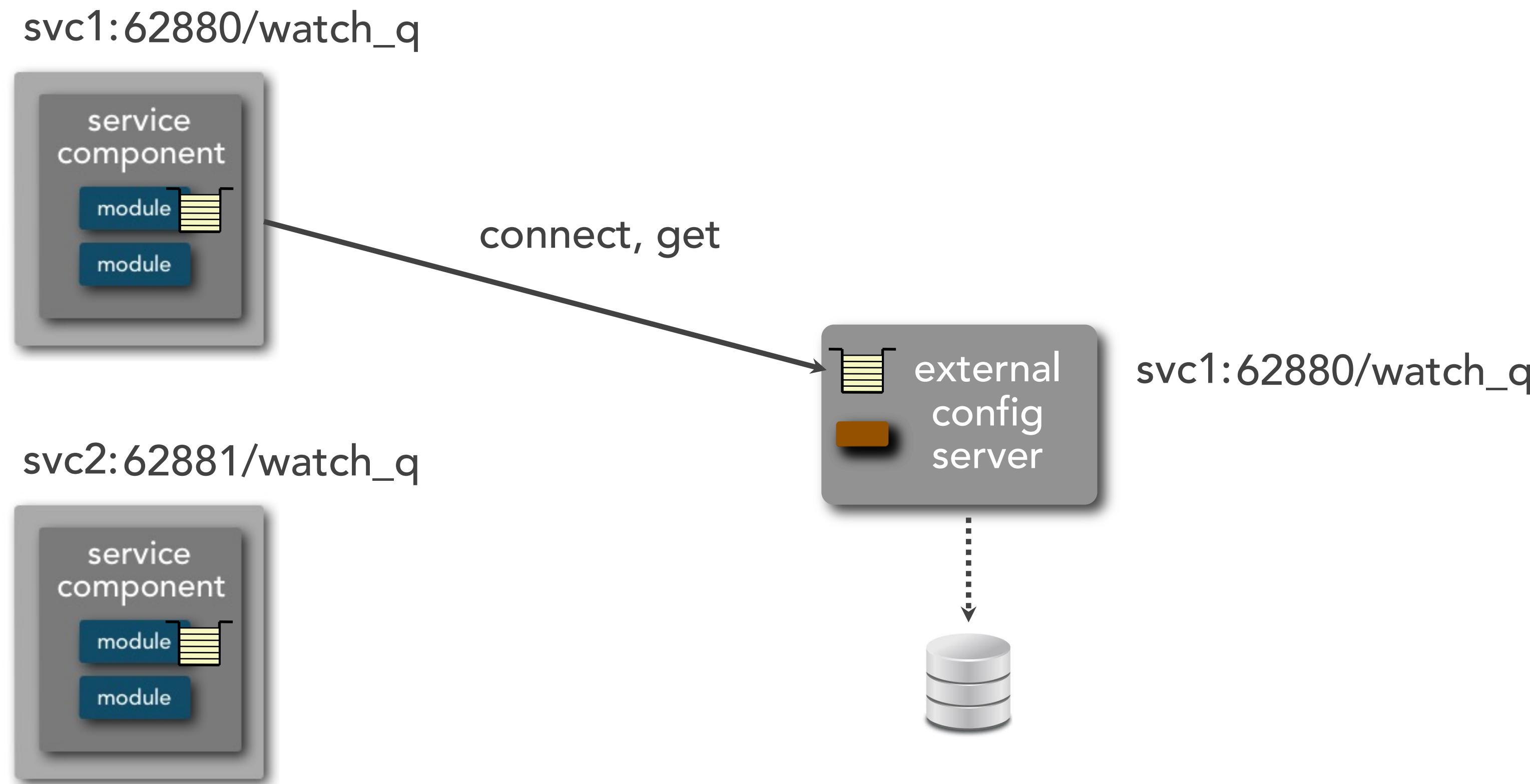
external config server



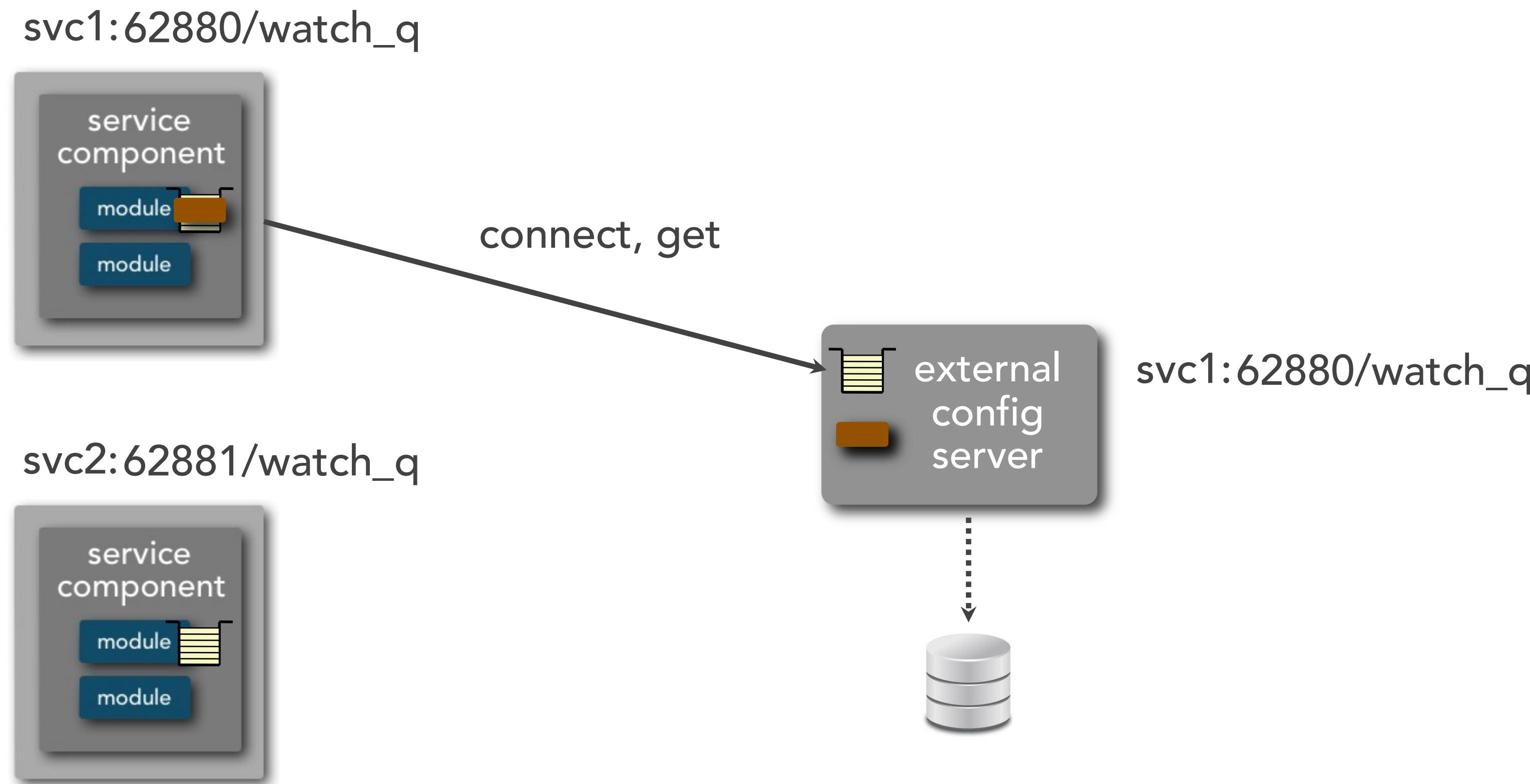
# watch notification pattern



# watch notification pattern

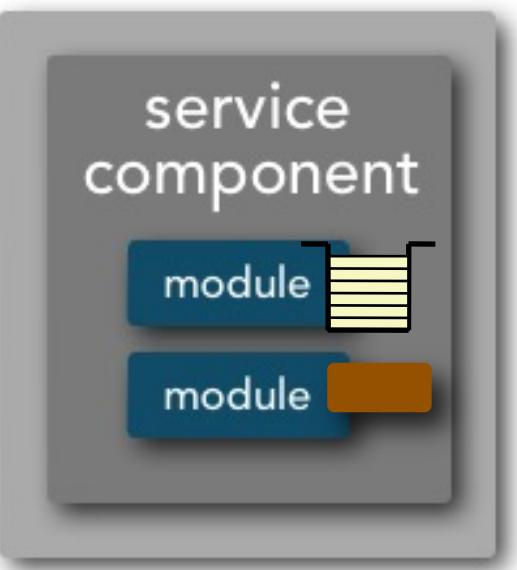


# watch notification pattern

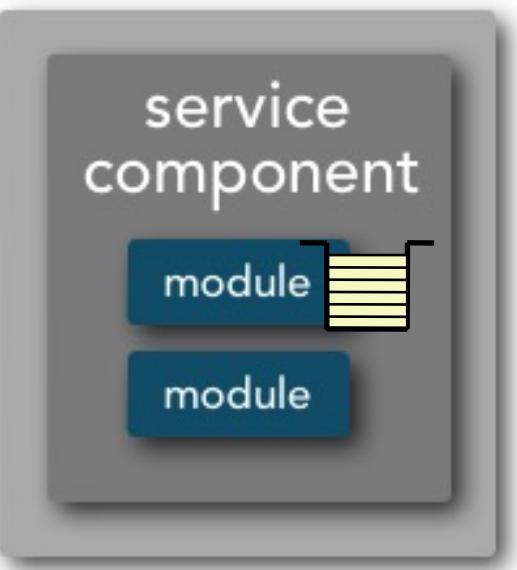


# watch notification pattern

svc1:62880/watch\_q

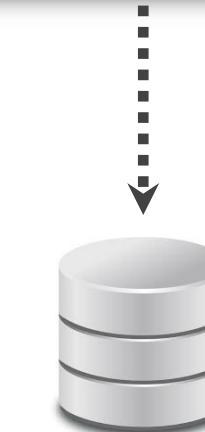


svc2:62881/watch\_q

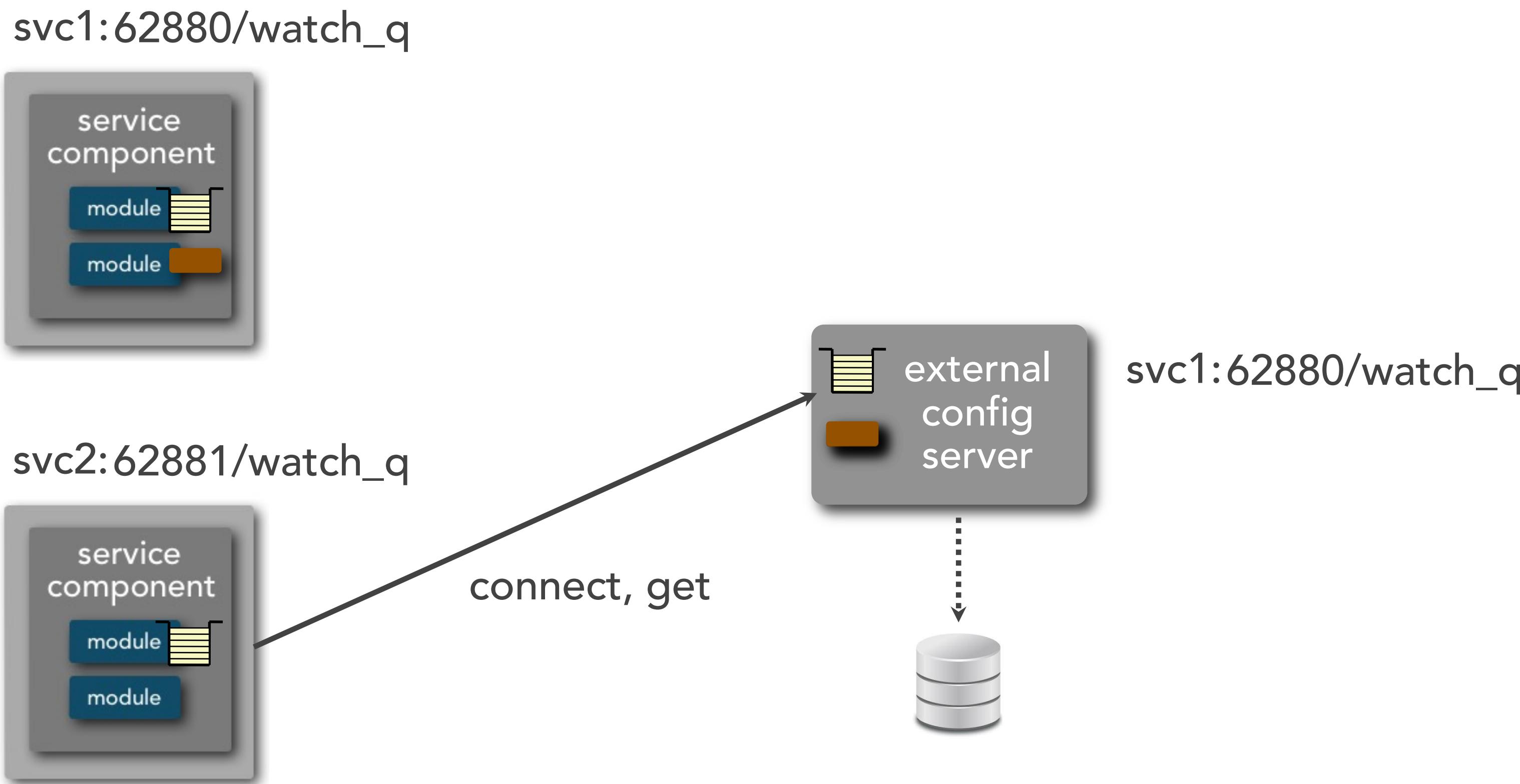


external config server

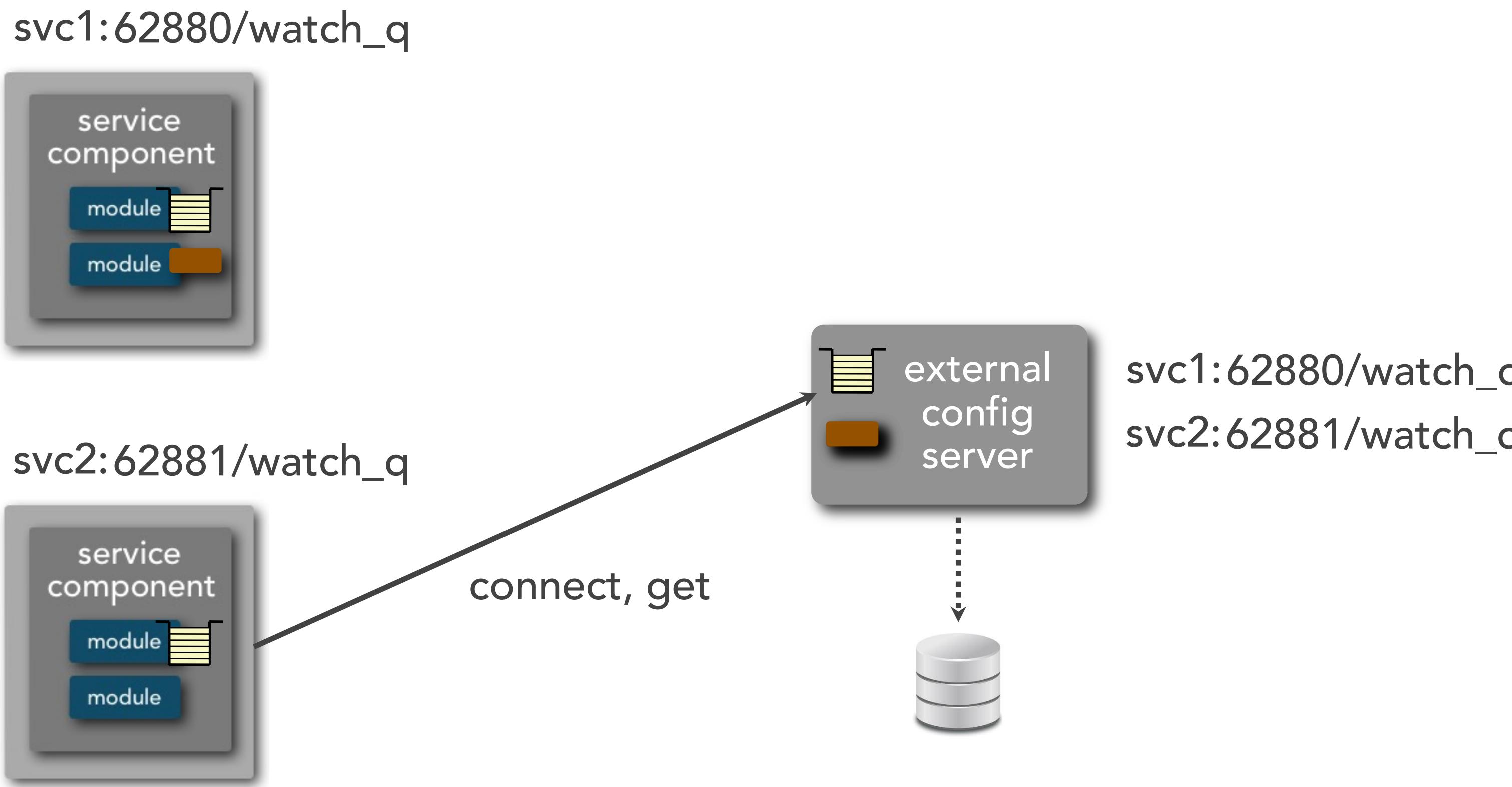
svc1:62880/watch\_q



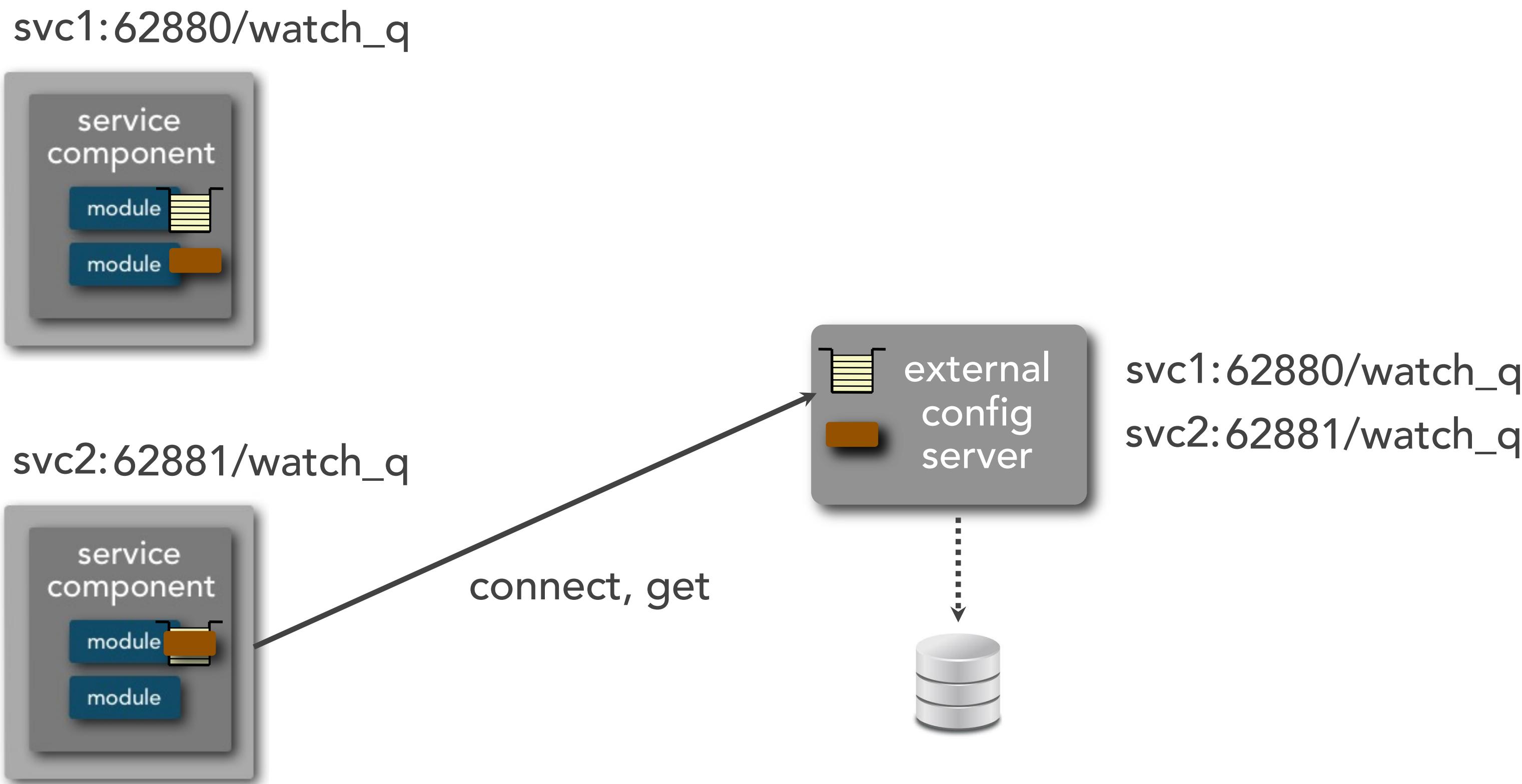
# watch notification pattern



# watch notification pattern

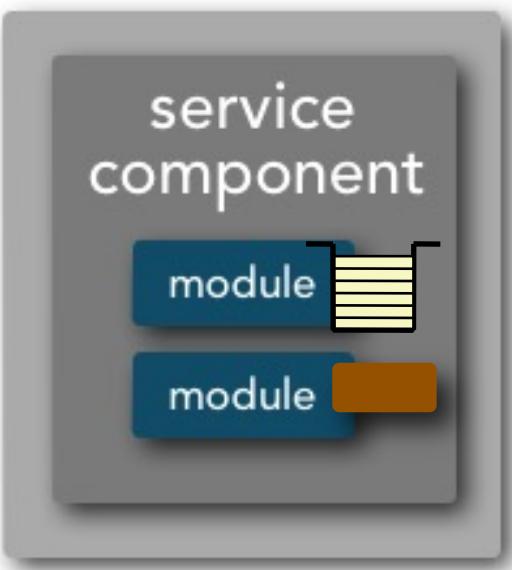


# watch notification pattern

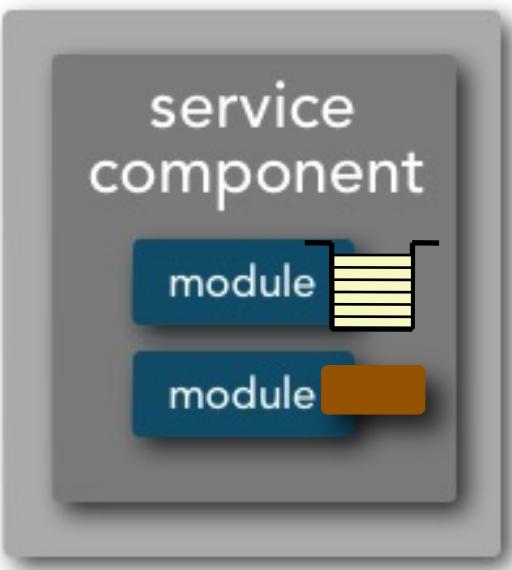


# watch notification pattern

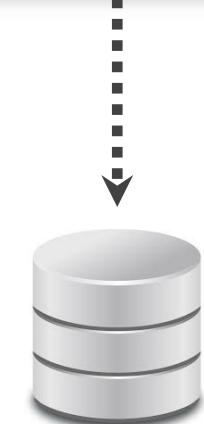
svc1:62880/watch\_q



svc2:62881/watch\_q



external config server

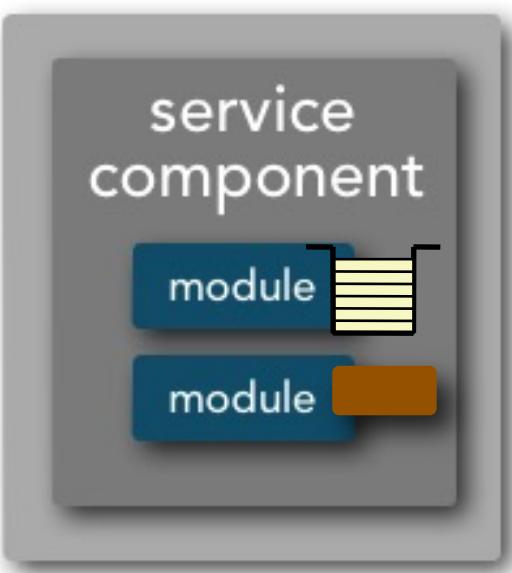


svc1:62880/watch\_q

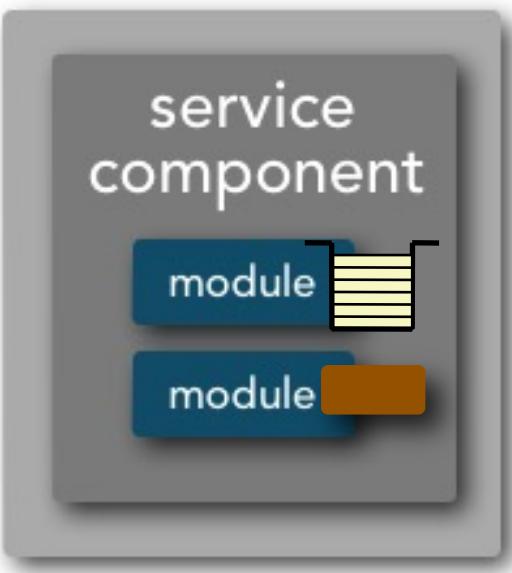
svc2:62881/watch\_q

# watch notification pattern

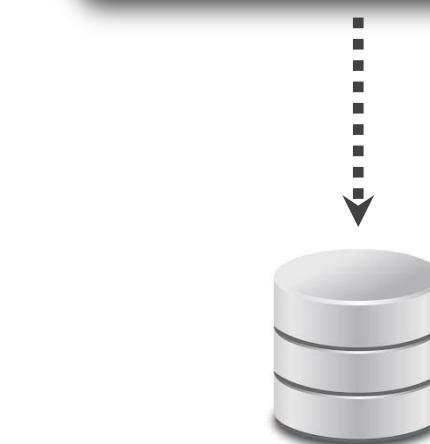
svc1:62880/watch\_q



svc2:62881/watch\_q



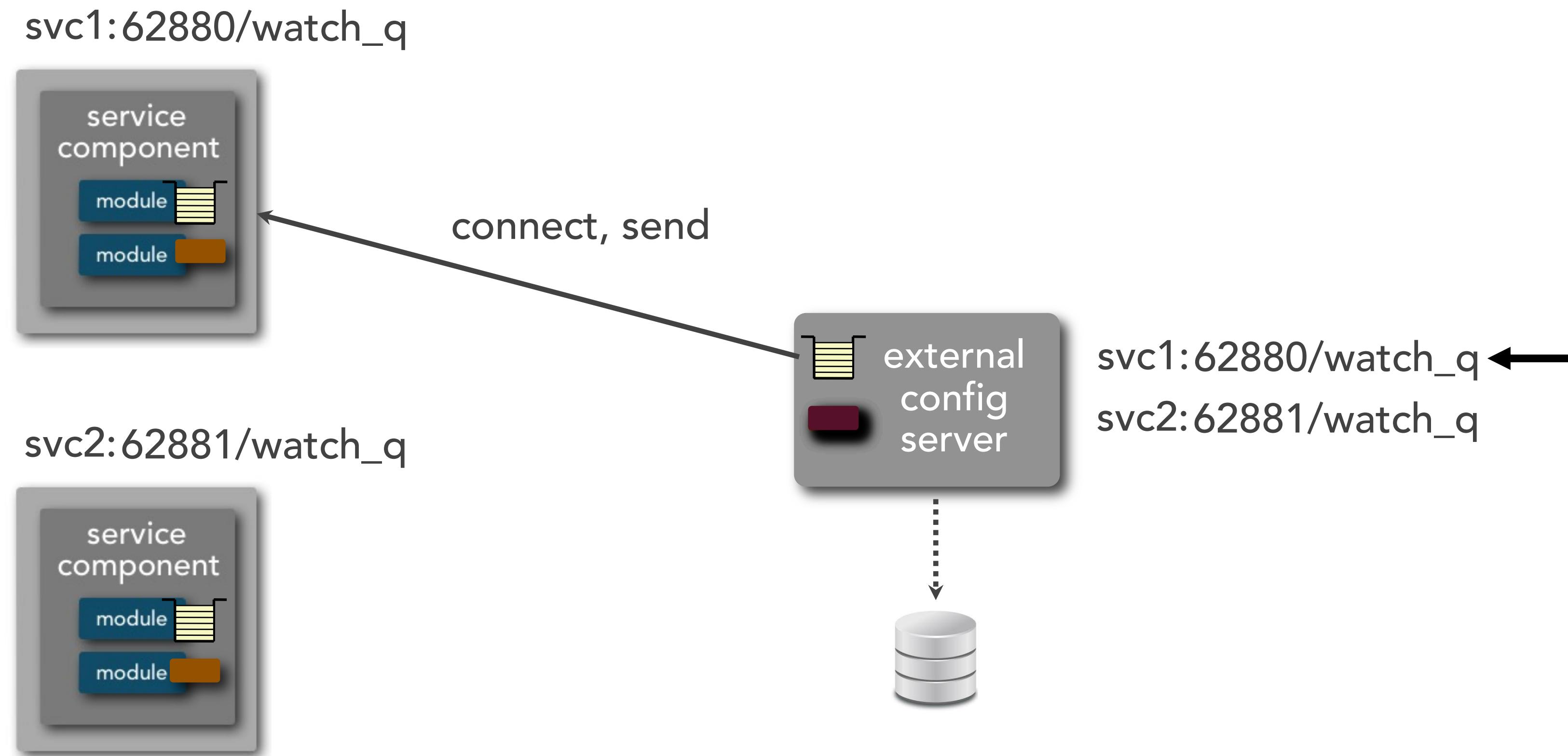
external config server



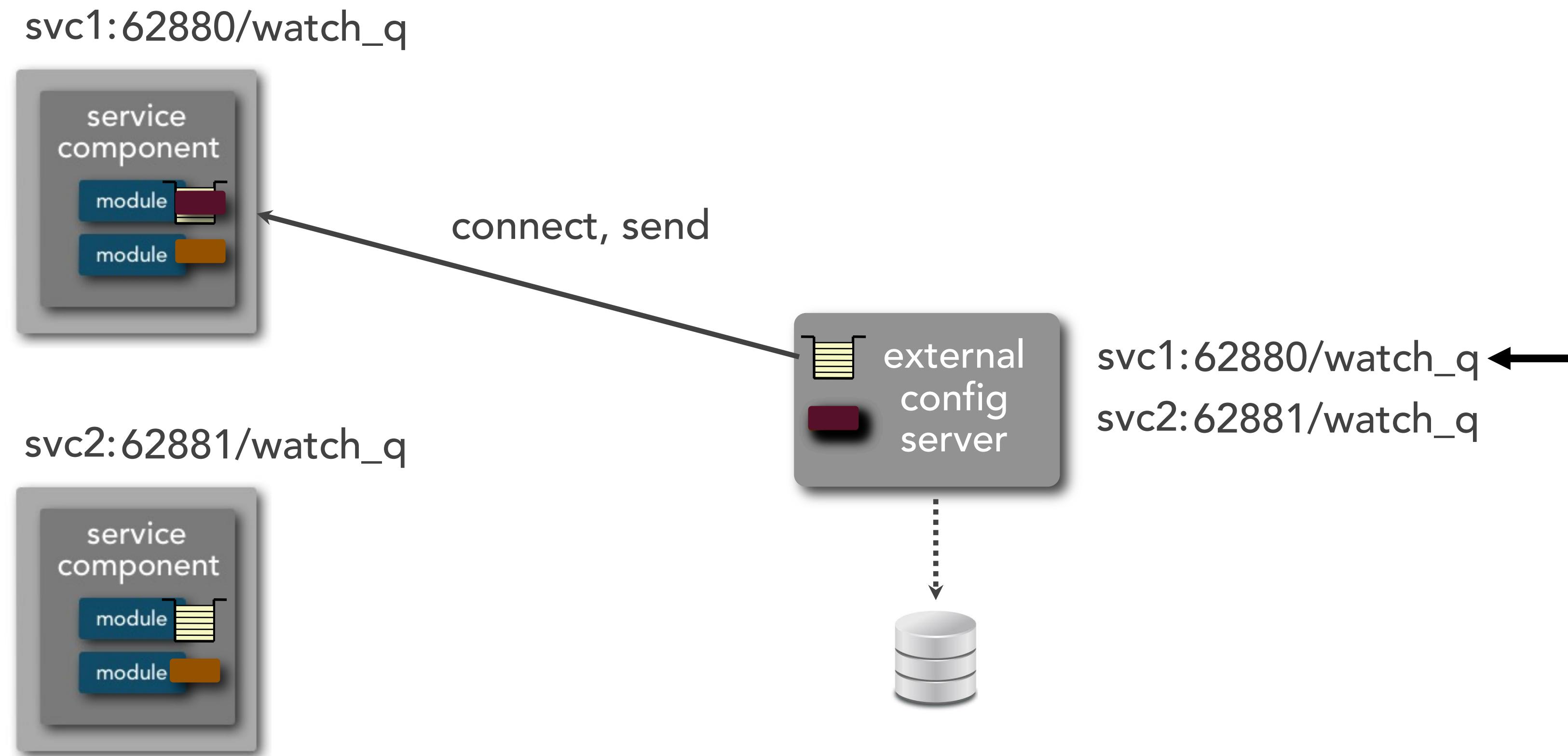
svc1:62880/watch\_q

svc2:62881/watch\_q

# watch notification pattern

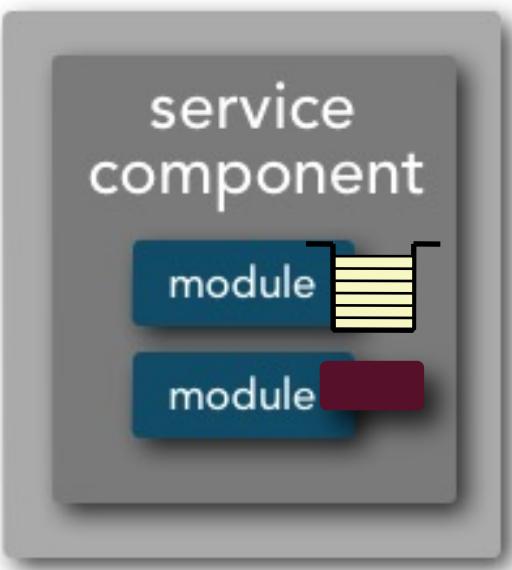


# watch notification pattern

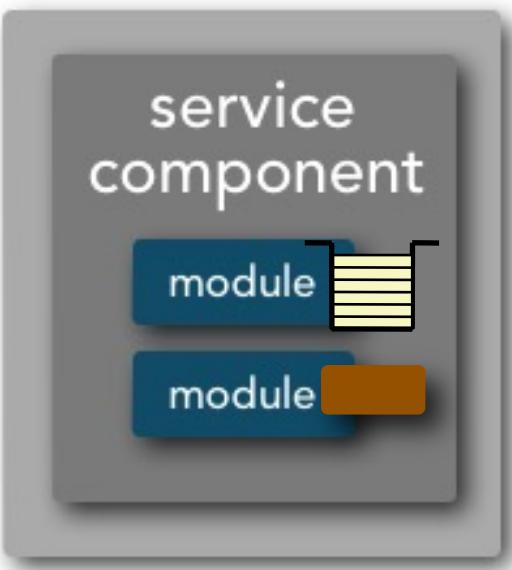


# watch notification pattern

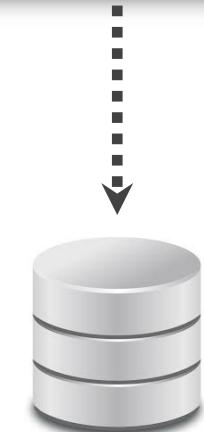
svc1:62880/watch\_q



svc2:62881/watch\_q

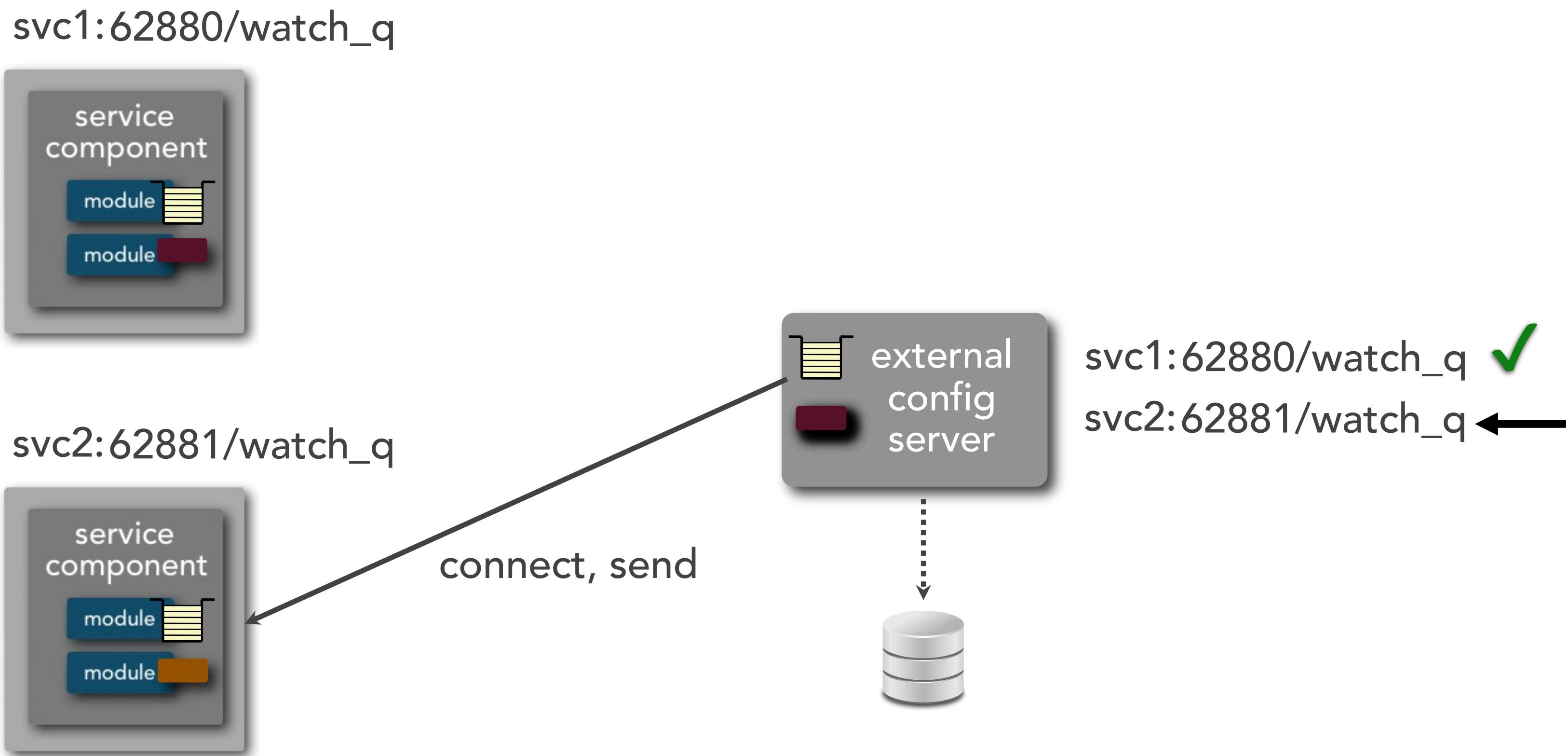


external config server

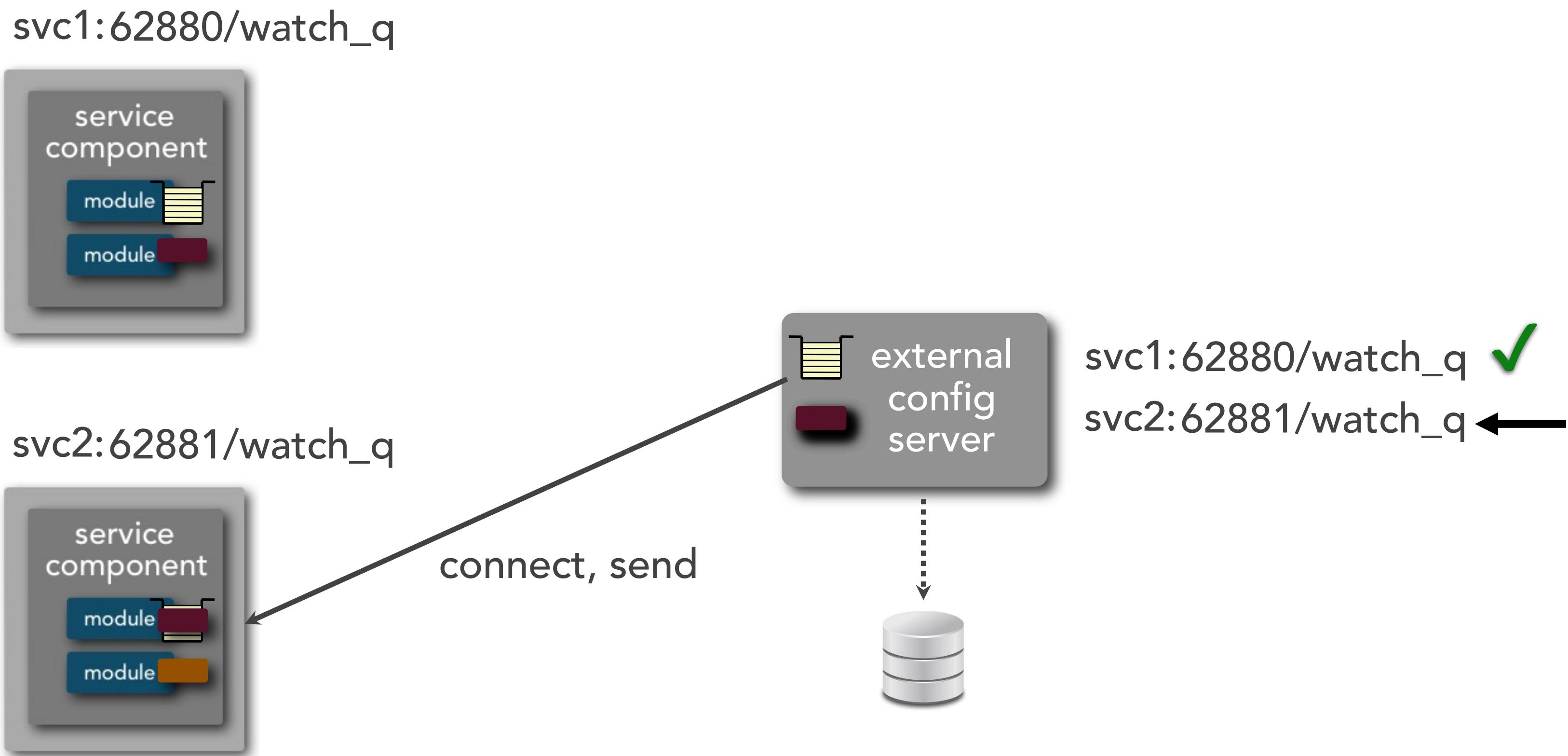


svc1:62880/watch\_q ✓  
svc2:62881/watch\_q

# watch notification pattern

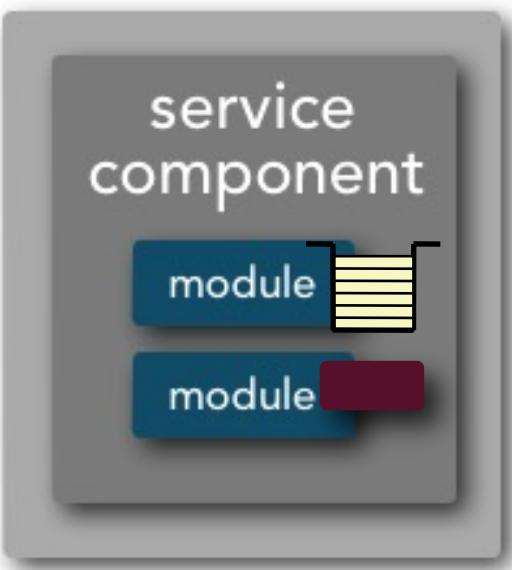


# watch notification pattern

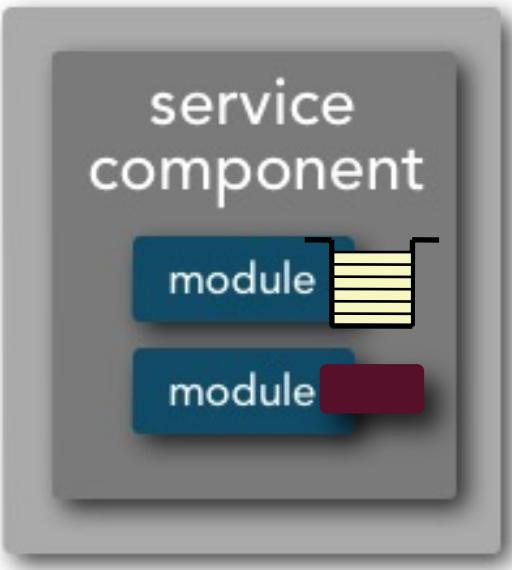


# watch notification pattern

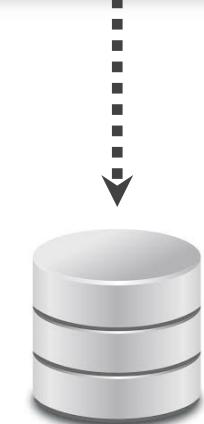
svc1:62880/watch\_q



svc2:62881/watch\_q



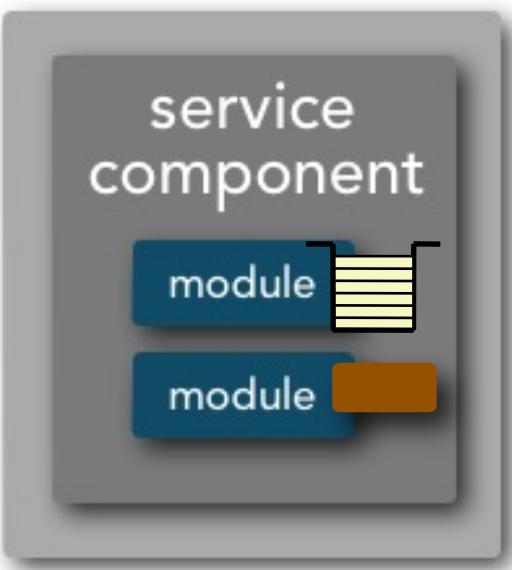
external config server



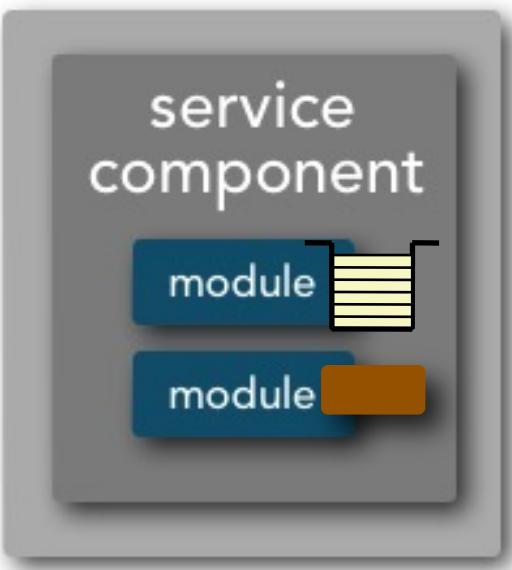
svc1:62880/watch\_q ✓  
svc2:62881/watch\_q ✓

# watch notification pattern

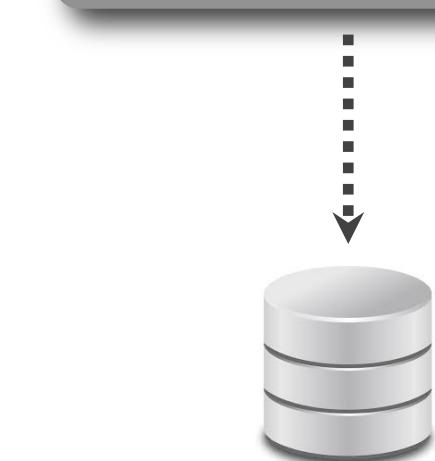
svc1:62880/watch\_q



svc2:62881/watch\_q



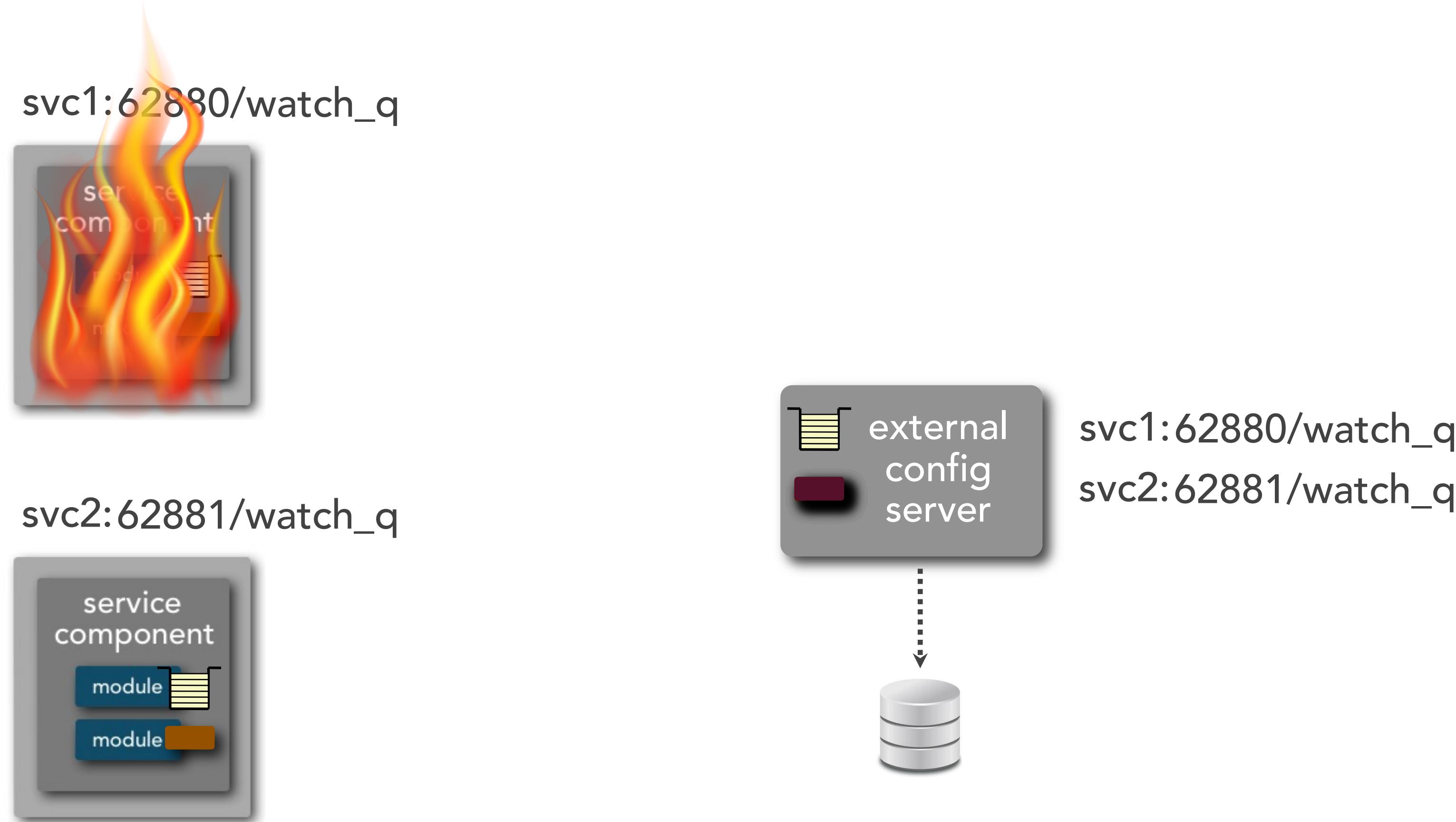
external config server



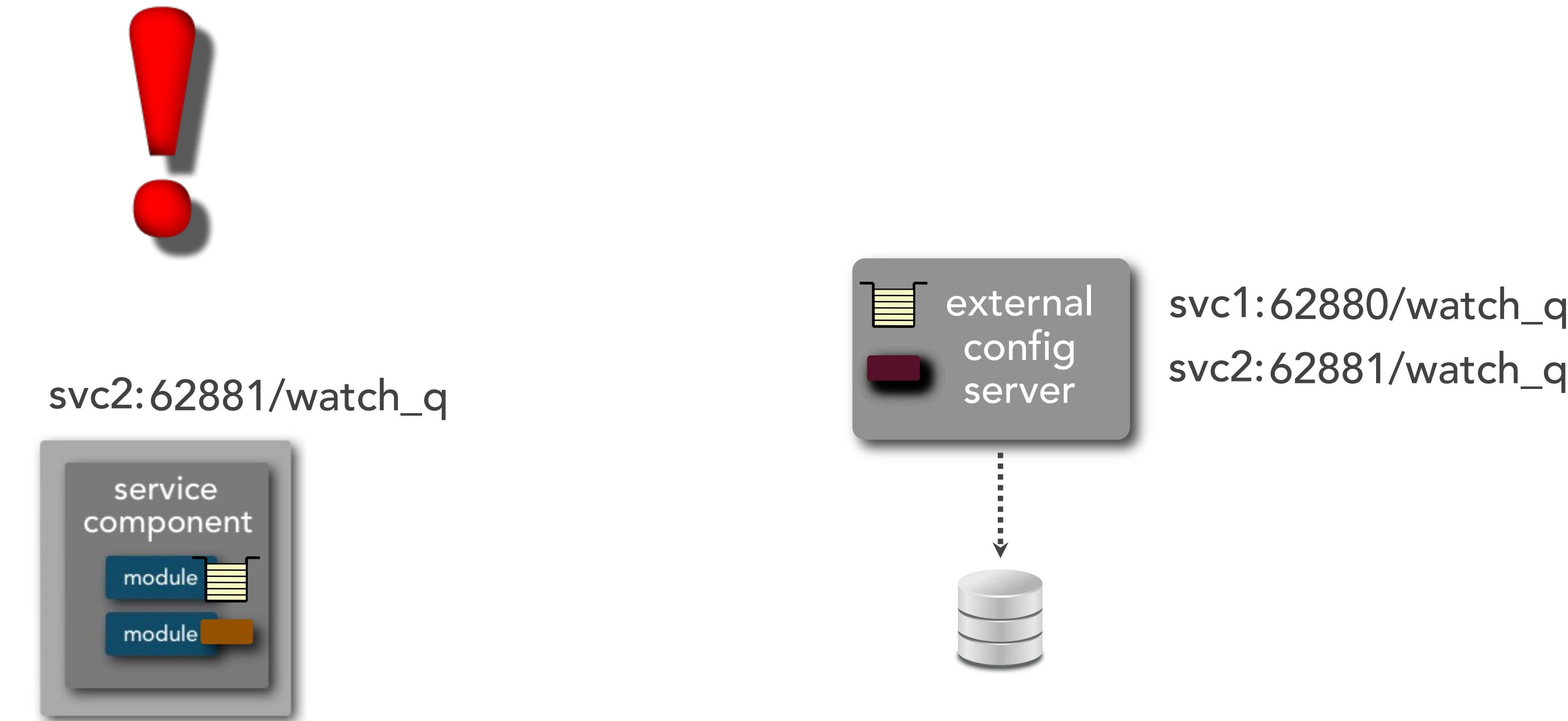
svc1:62880/watch\_q

svc2:62881/watch\_q

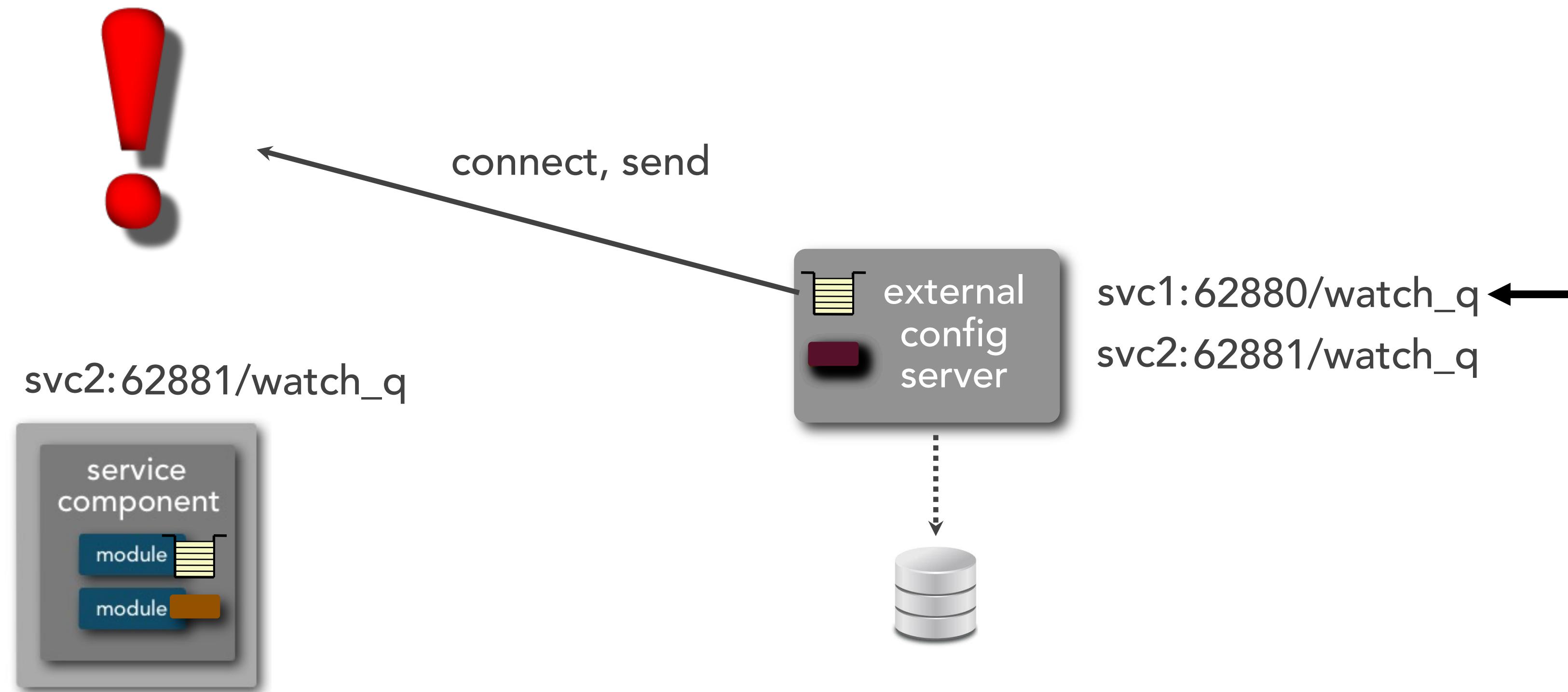
# watch notification pattern



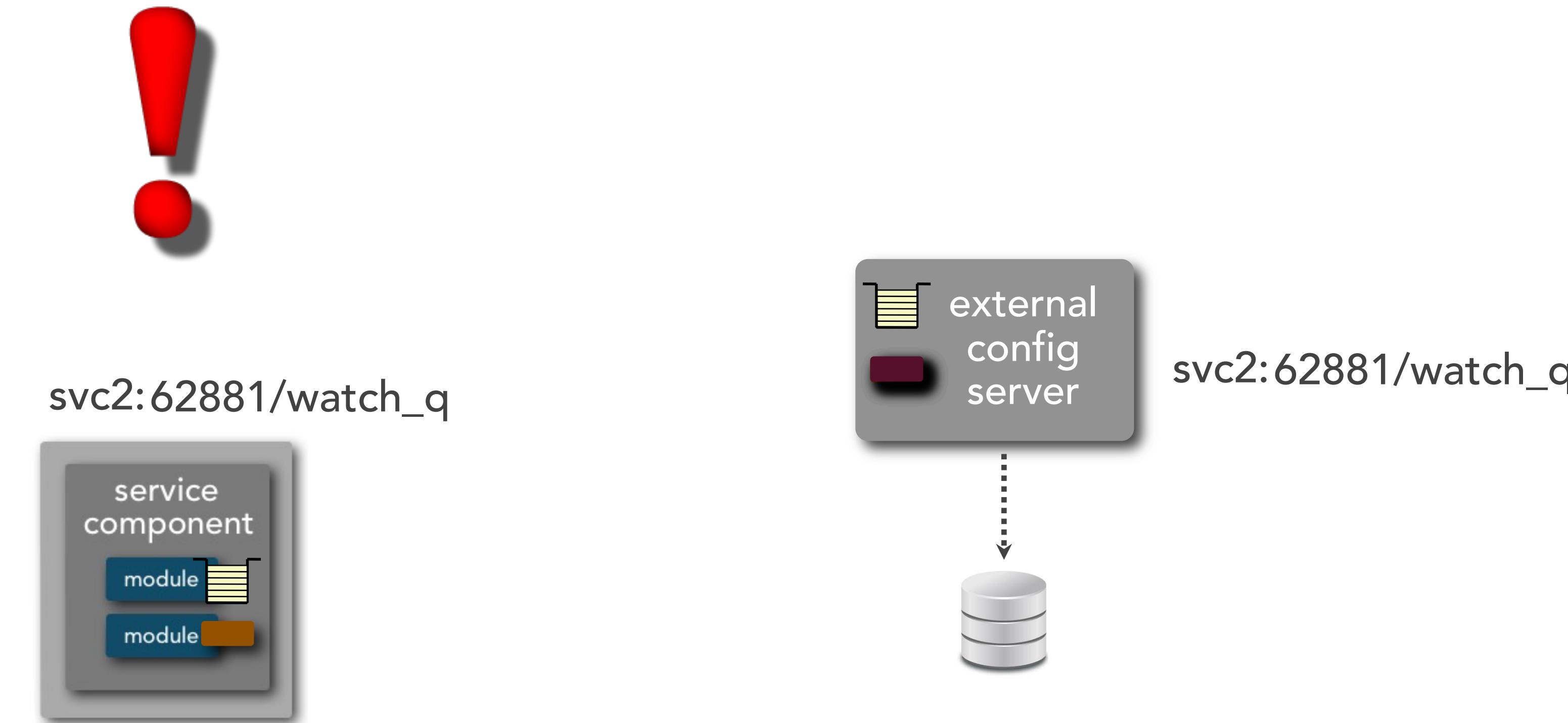
# watch notification pattern



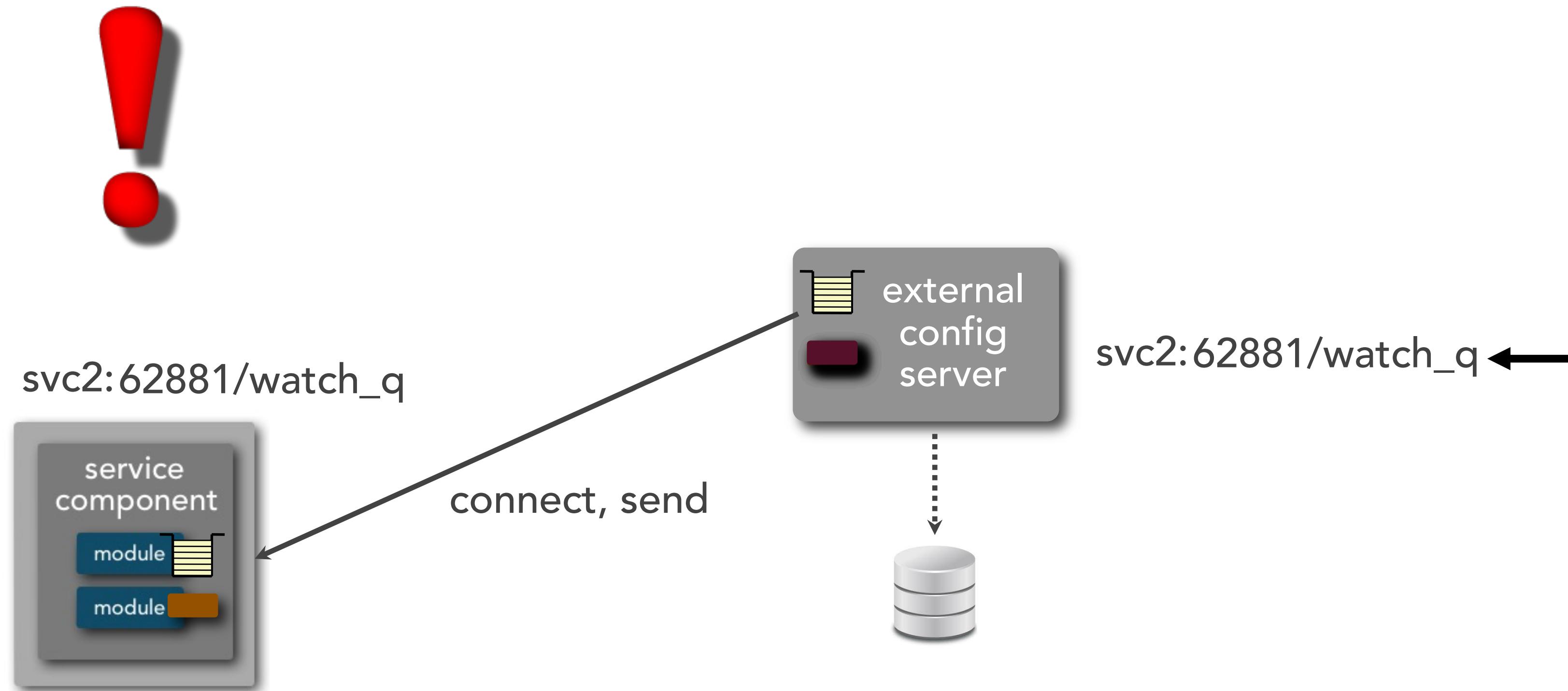
# watch notification pattern



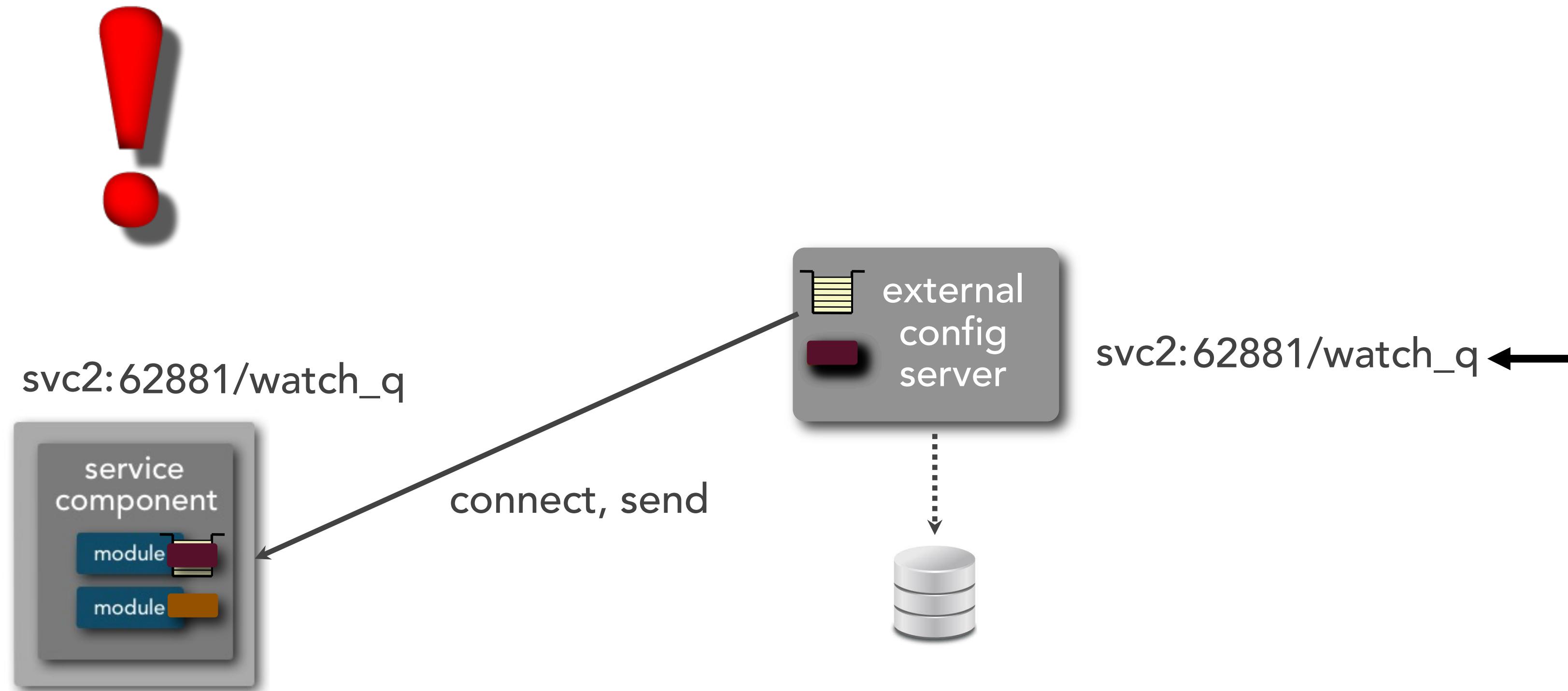
# watch notification pattern



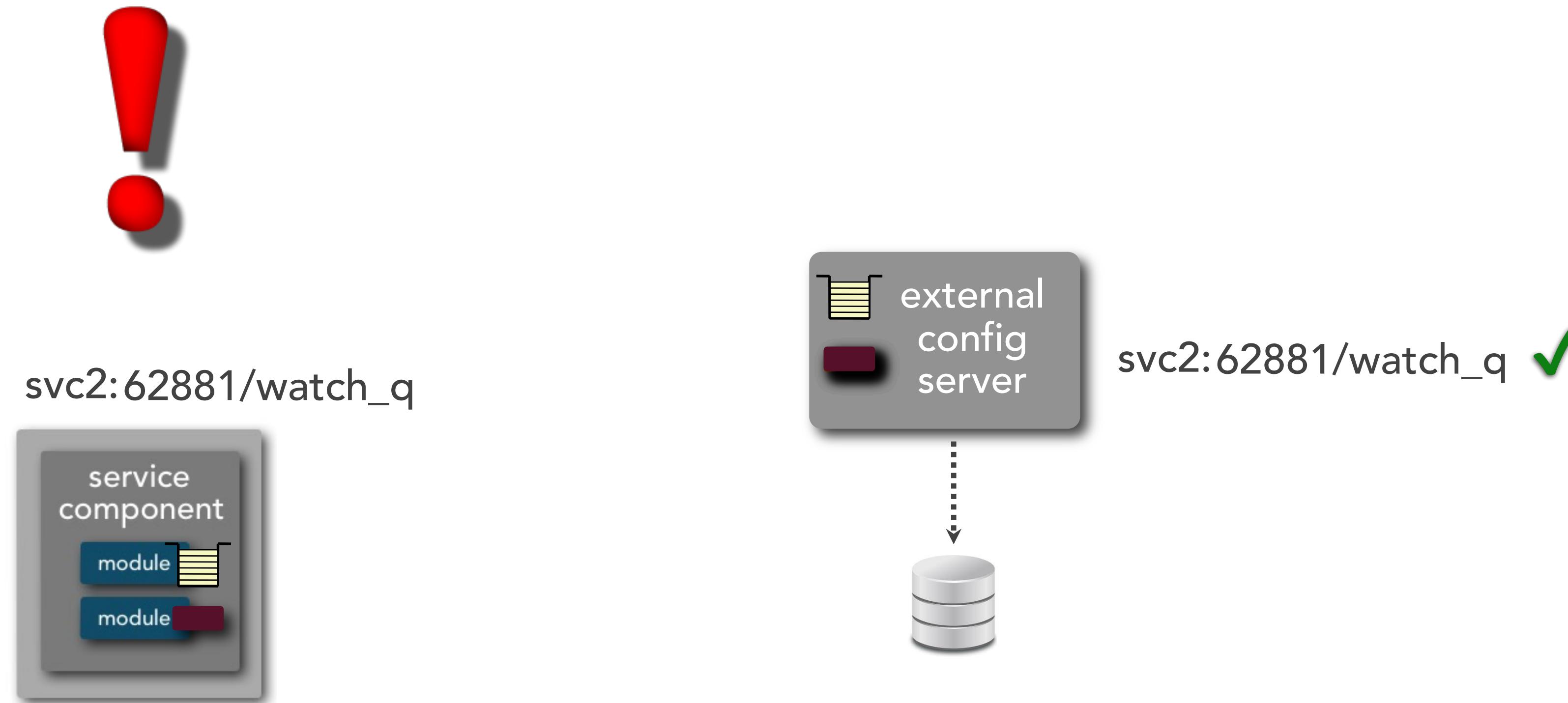
# watch notification pattern



# watch notification pattern

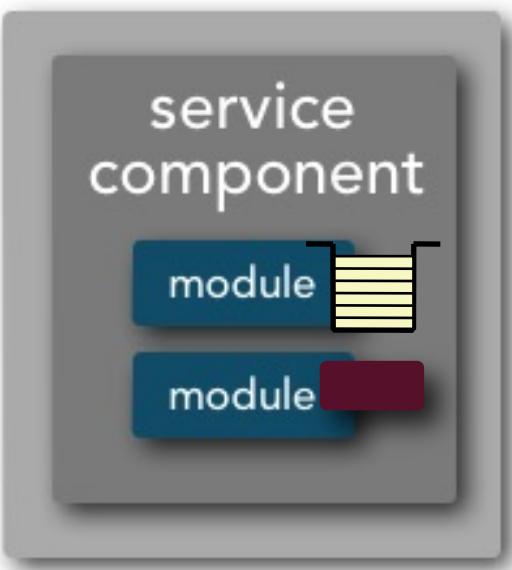


# watch notification pattern

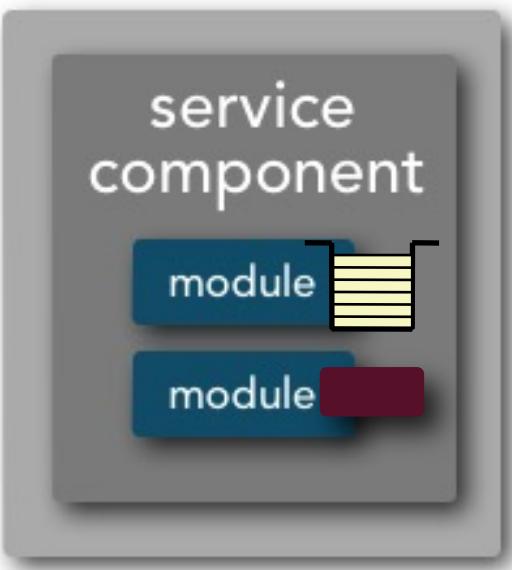


# watch notification pattern

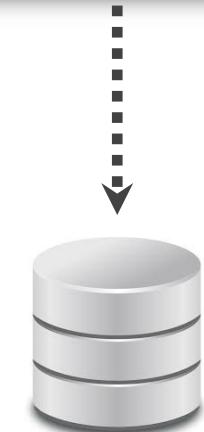
svc1:62880/watch\_q



svc2:62881/watch\_q



external config server



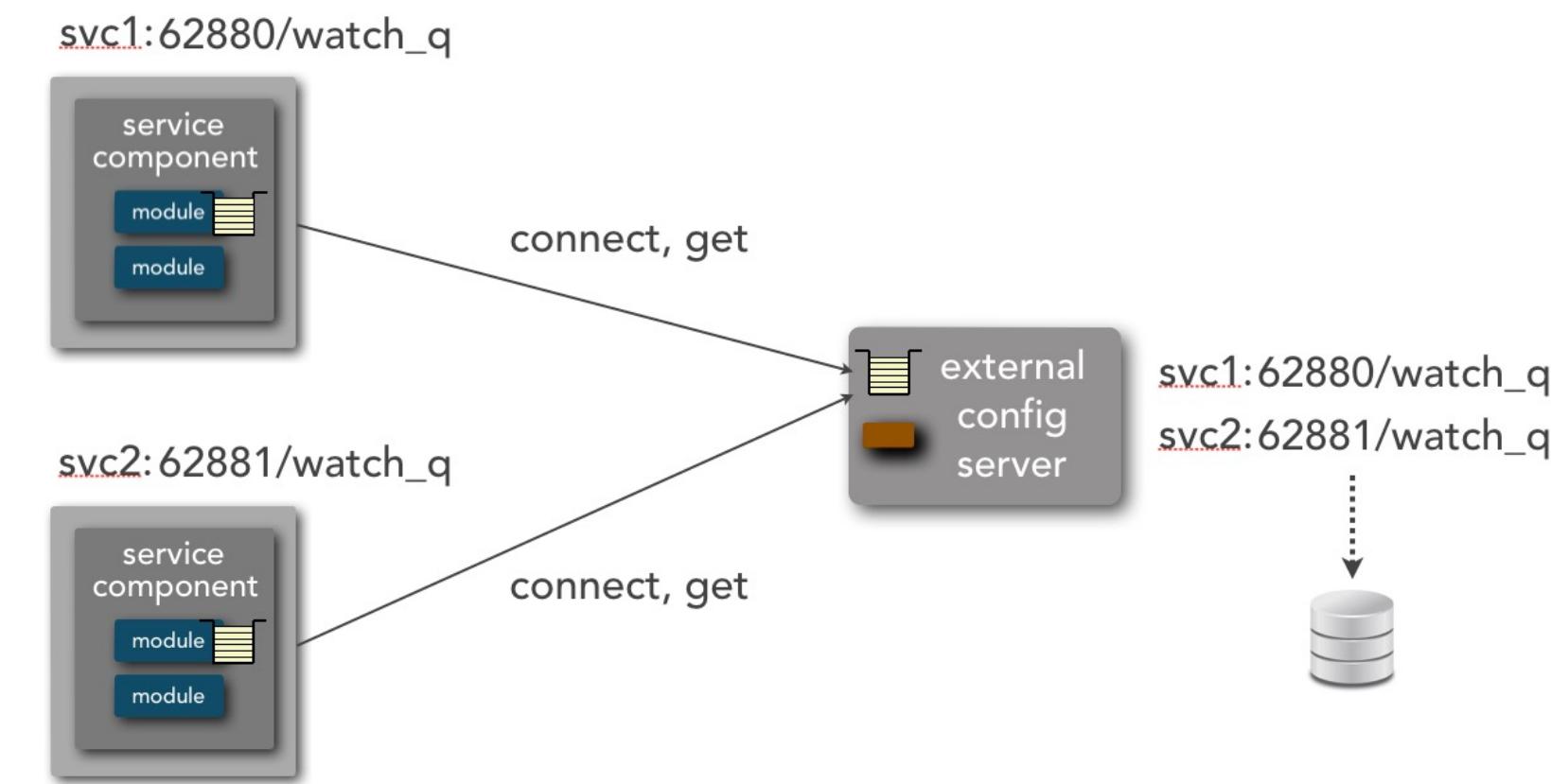
svc1:62880/watch\_q ✓  
svc2:62881/watch\_q ✓

# watch notification pattern

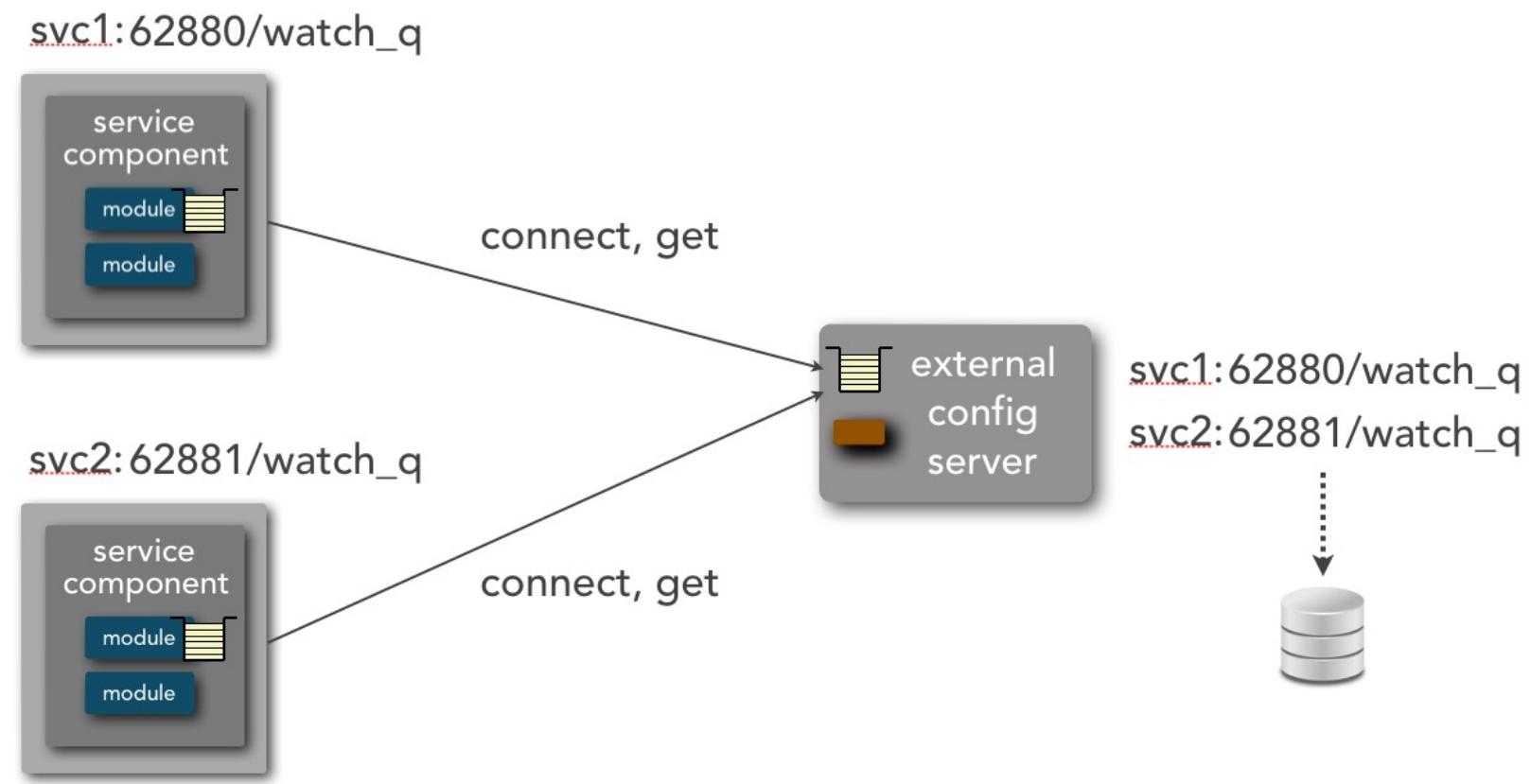


let's apply the pattern...

# watch notification pattern



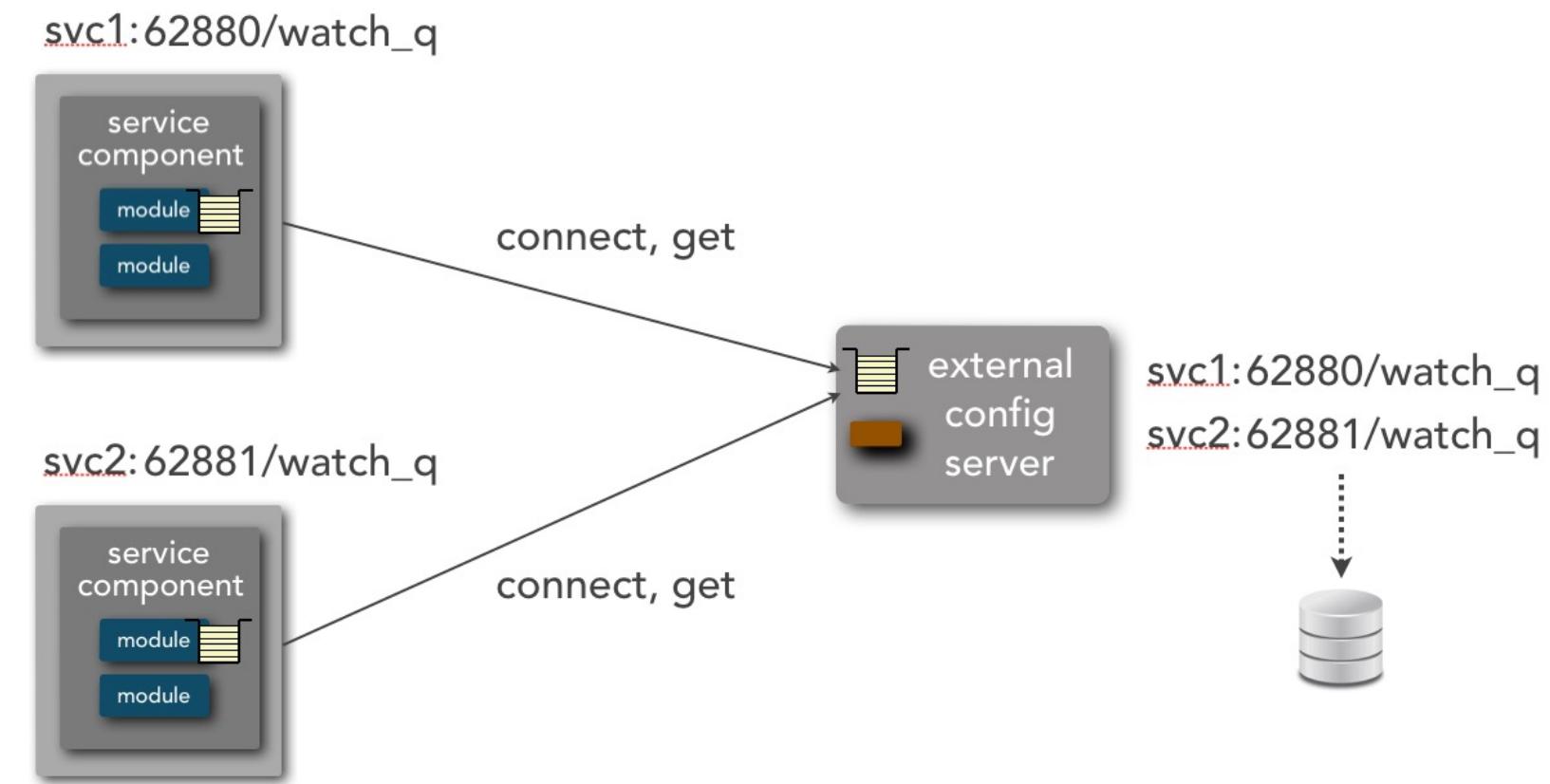
# watch notification pattern



lower bandwidth  
fewer connections  
fault tolerance



# watch notification pattern



lower bandwidth  
fewer connections  
fault tolerance



performance  
complexity  
error handing

# Supervisor Consumer Pattern

# supervisor consumer pattern

GitHub



[https://github.com/wmr513/reactive/tree/master/  
supervisor](https://github.com/wmr513/reactive/tree/master/supervisor)

# supervisor consumer pattern

*“how do I handle varying request load and still maintain a consistent response time?”*

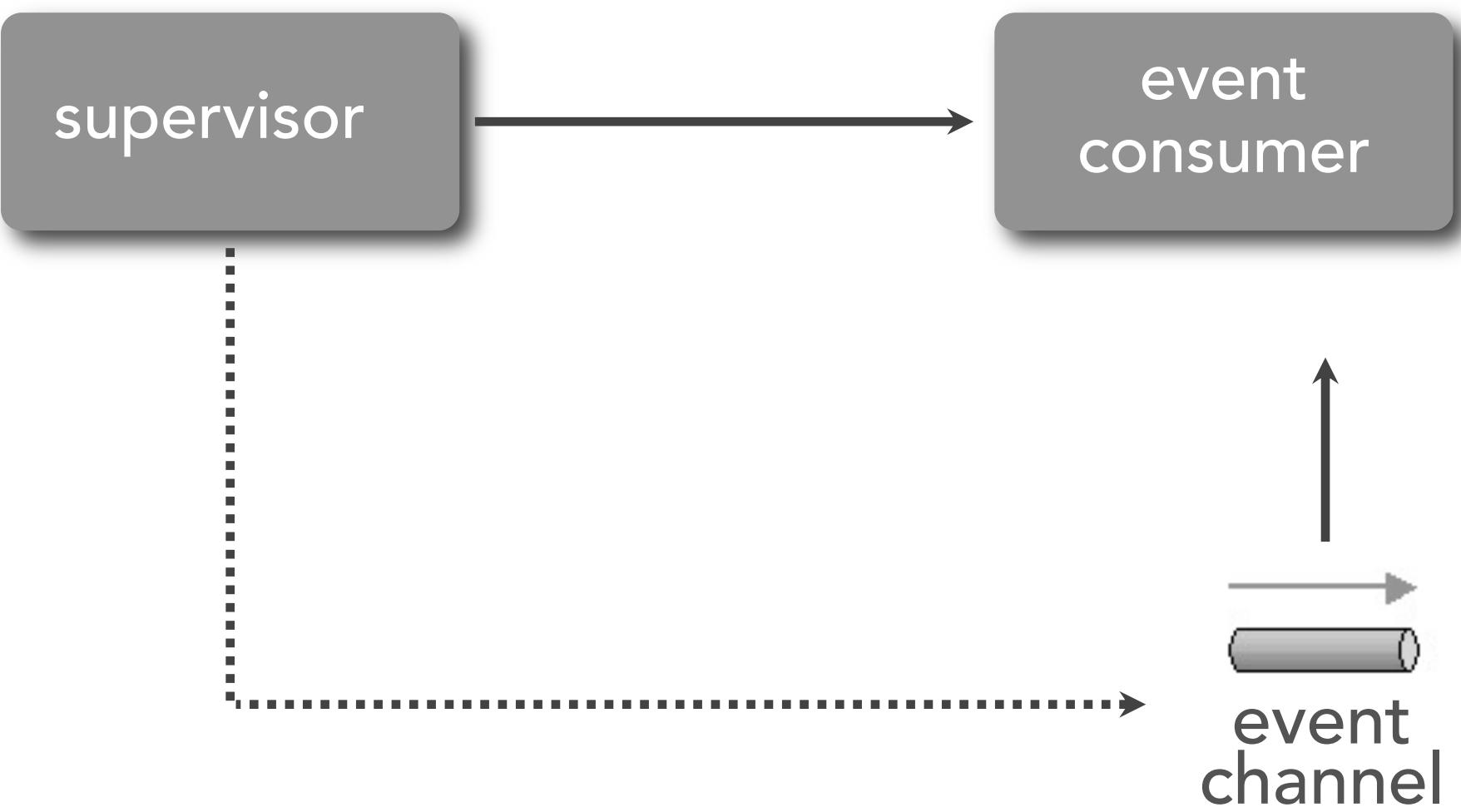


# supervisor consumer pattern

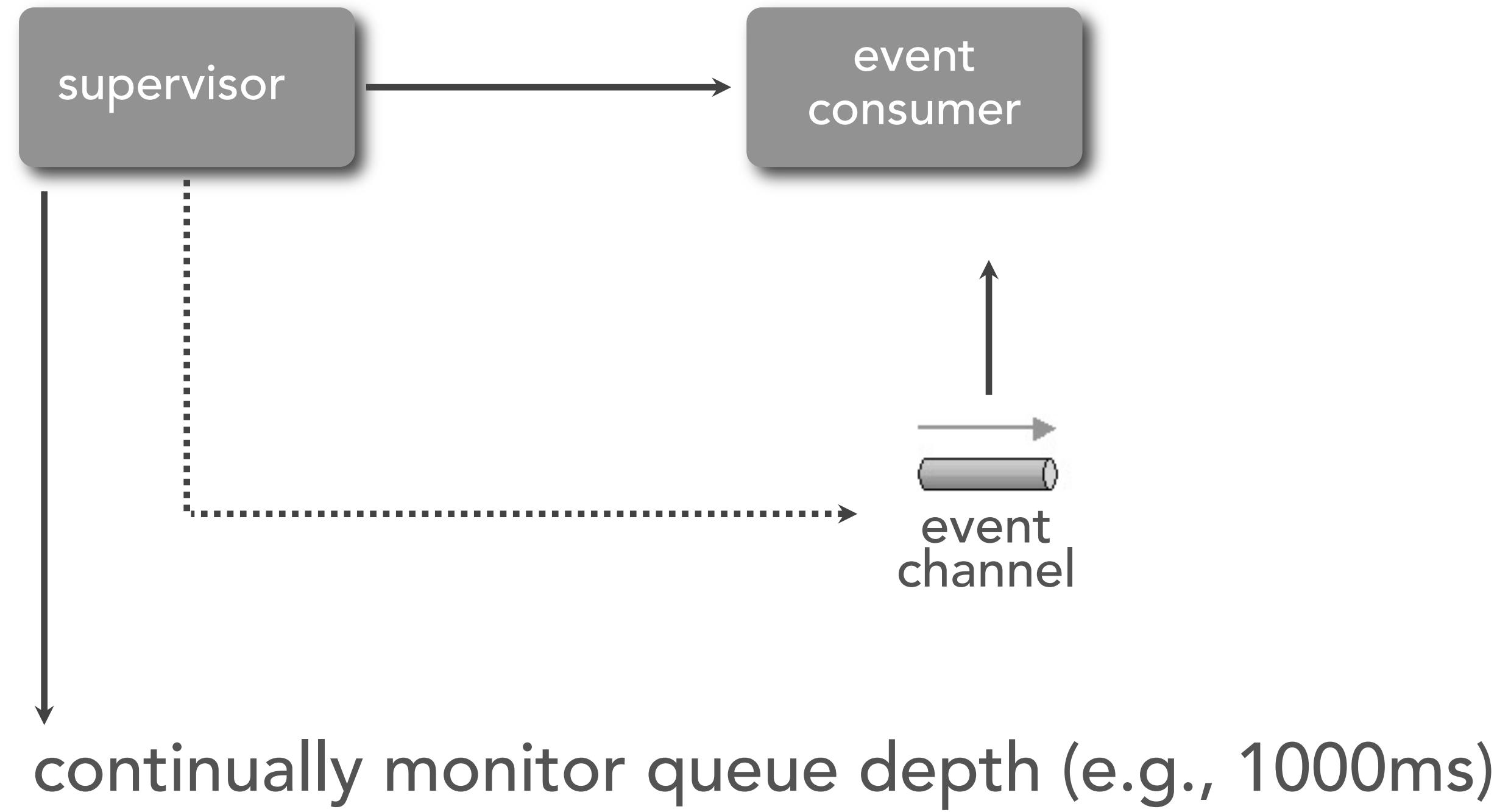


let's see the issue...

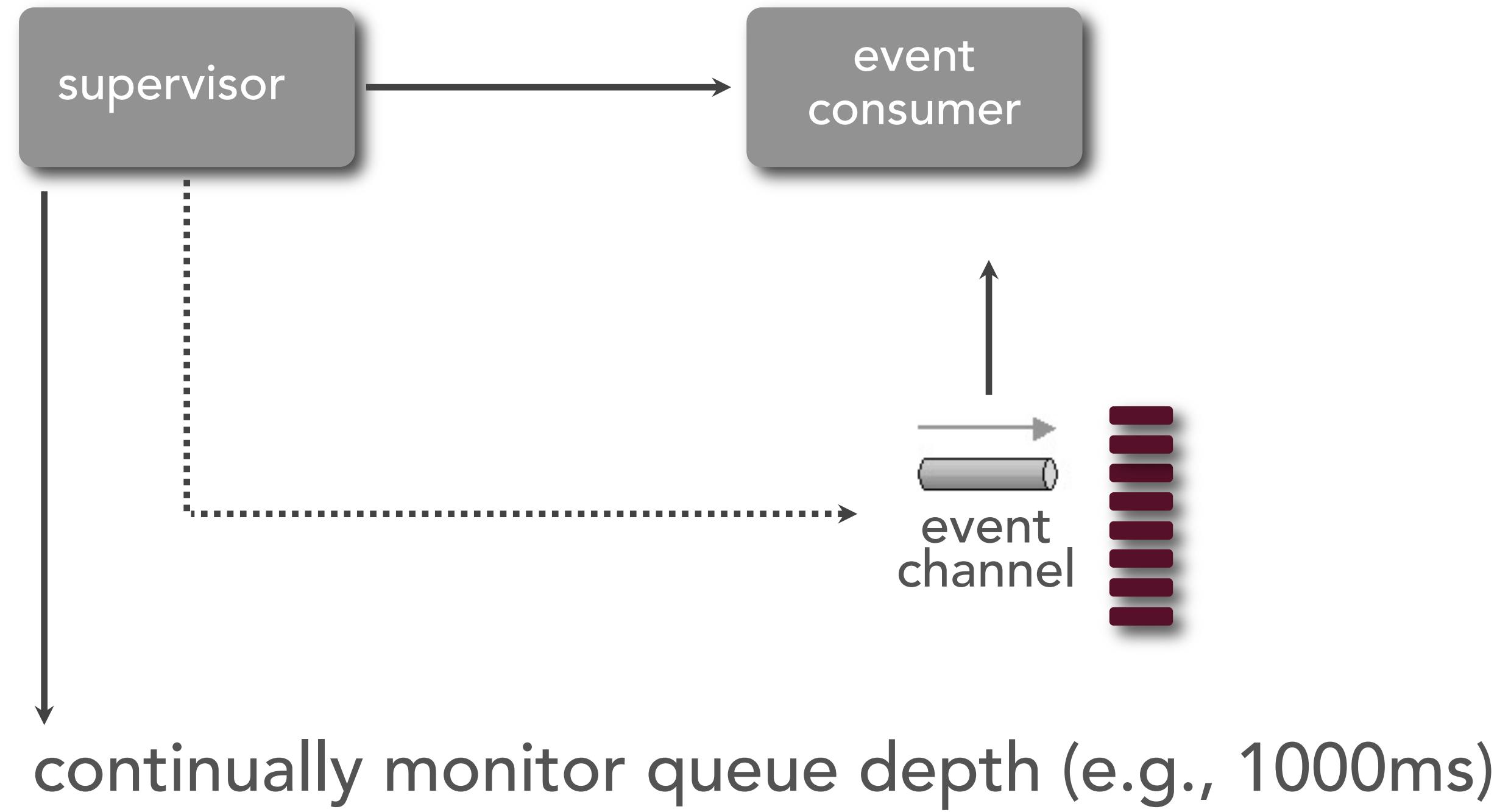
# supervisor consumer pattern



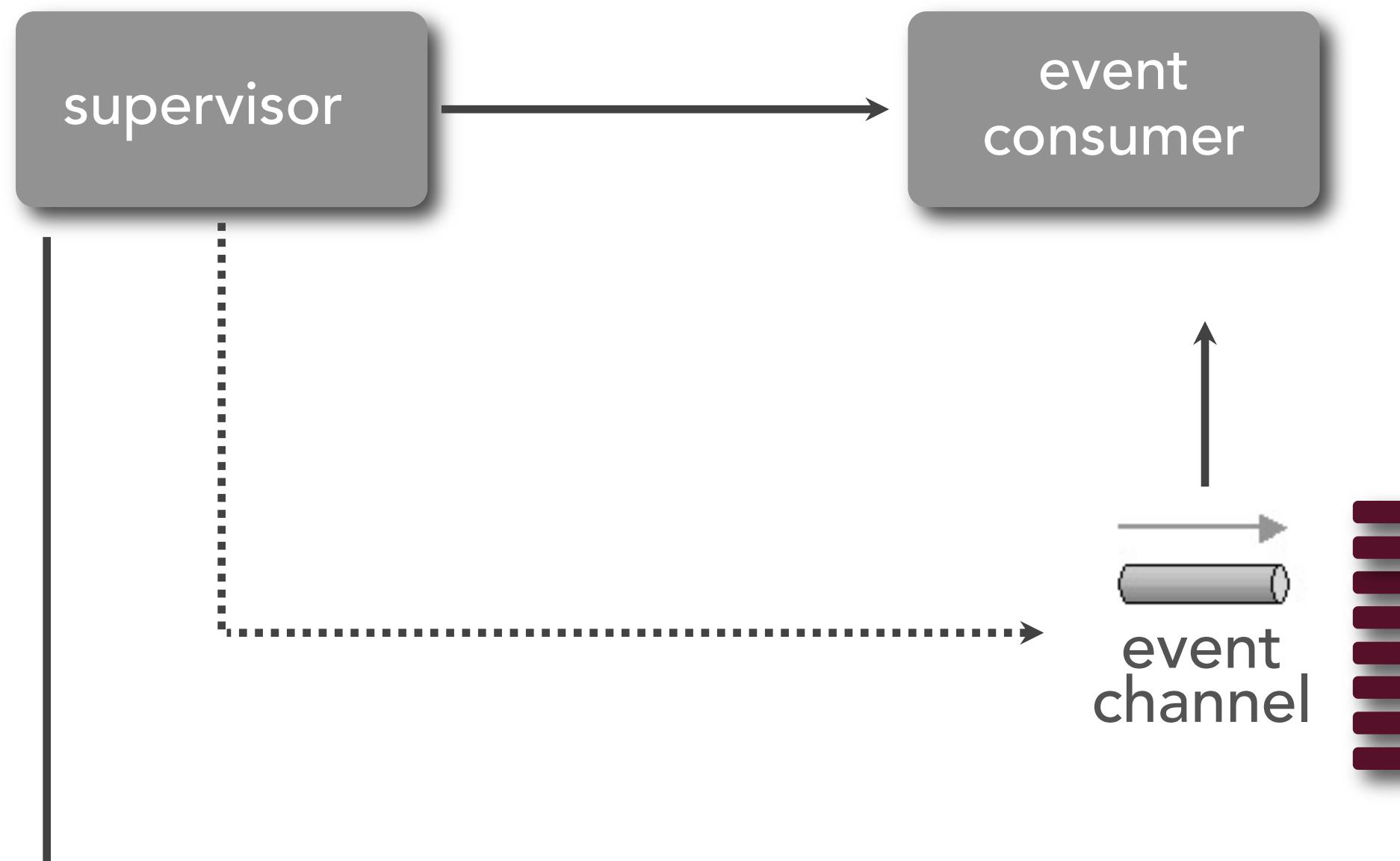
# supervisor consumer pattern



# supervisor consumer pattern

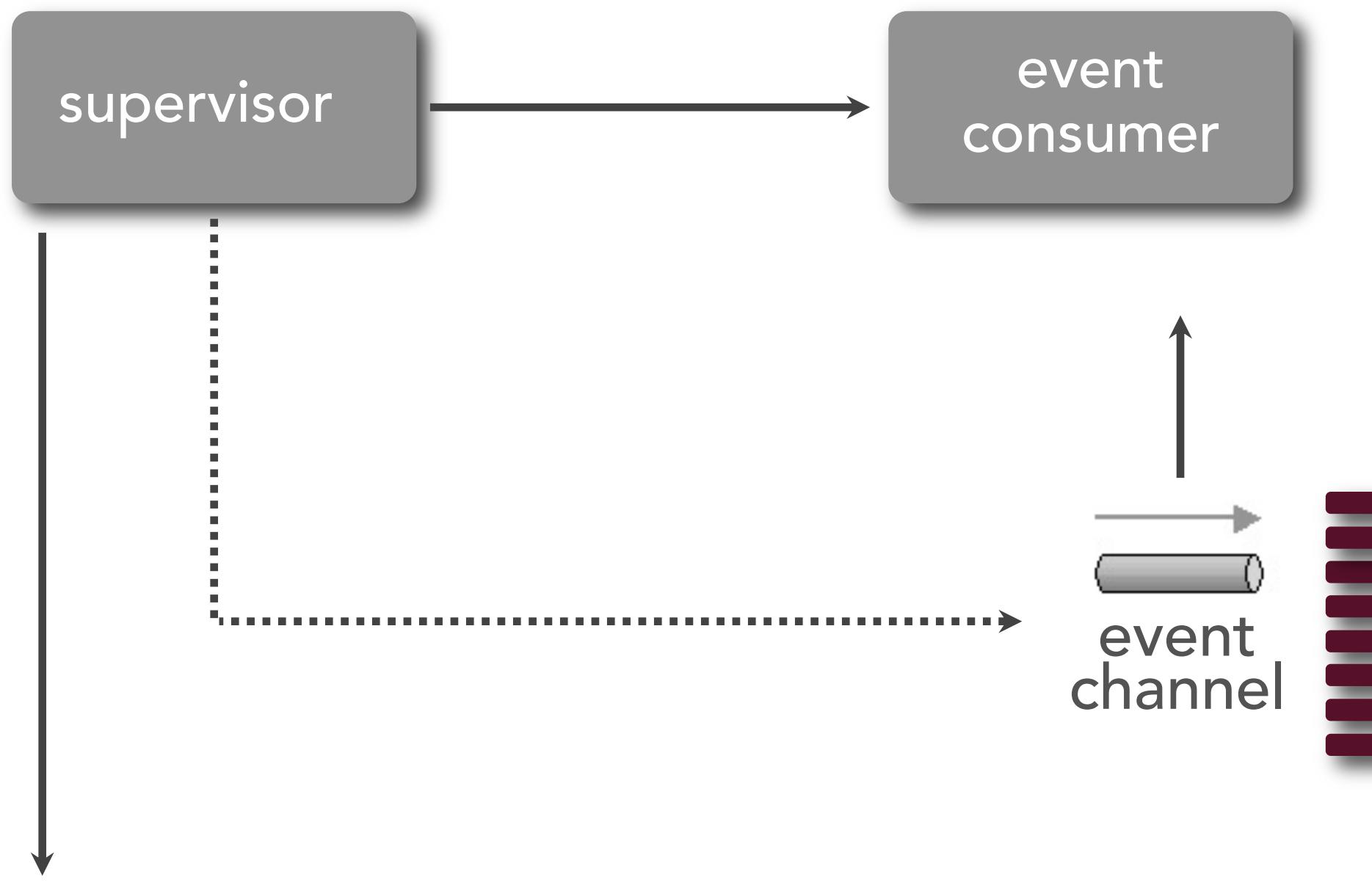


# supervisor consumer pattern



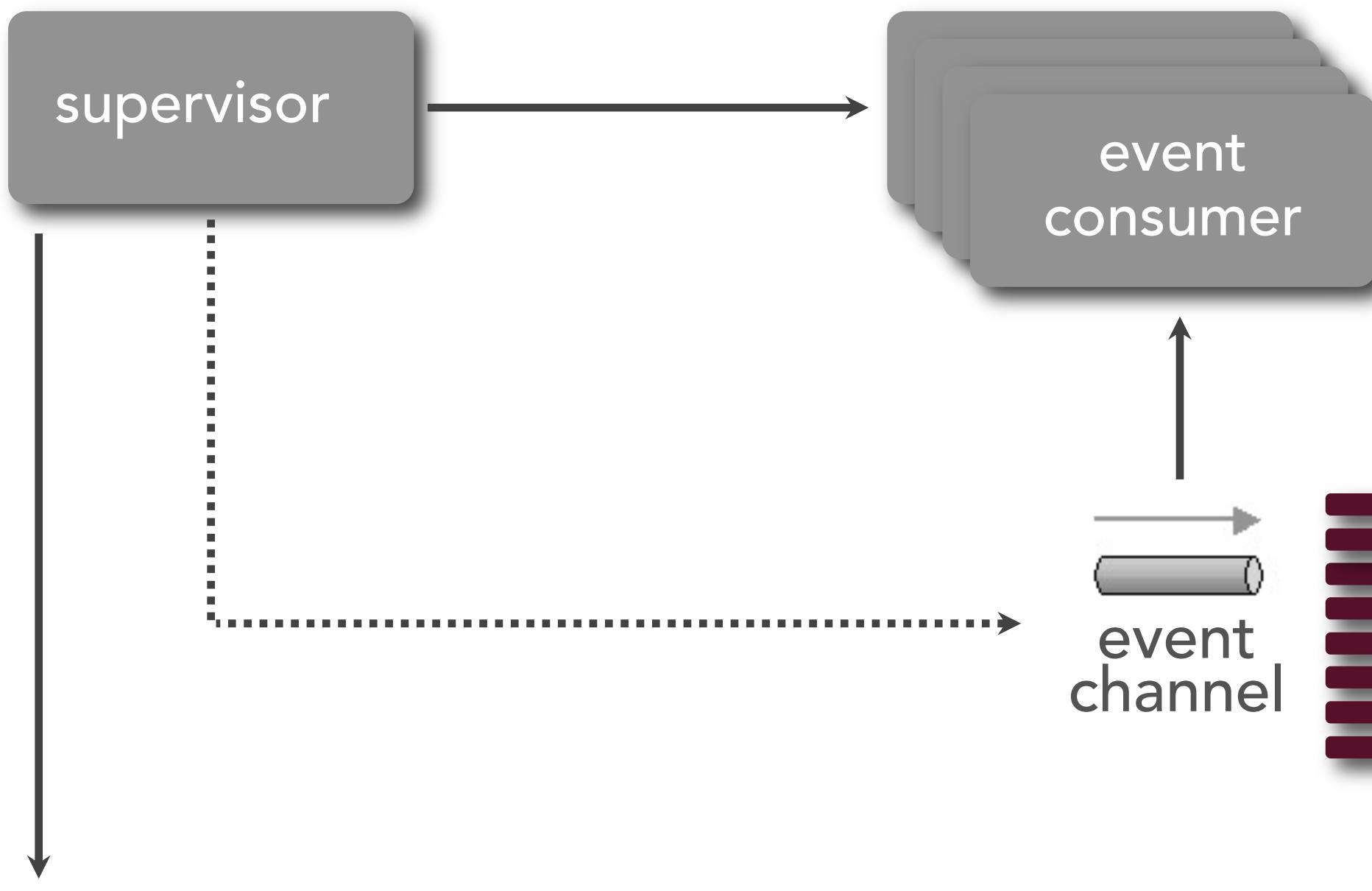
continually monitor queue depth (e.g., 1000ms)  
determine consumers needed ( $\text{depth}/n$ )

# supervisor consumer pattern



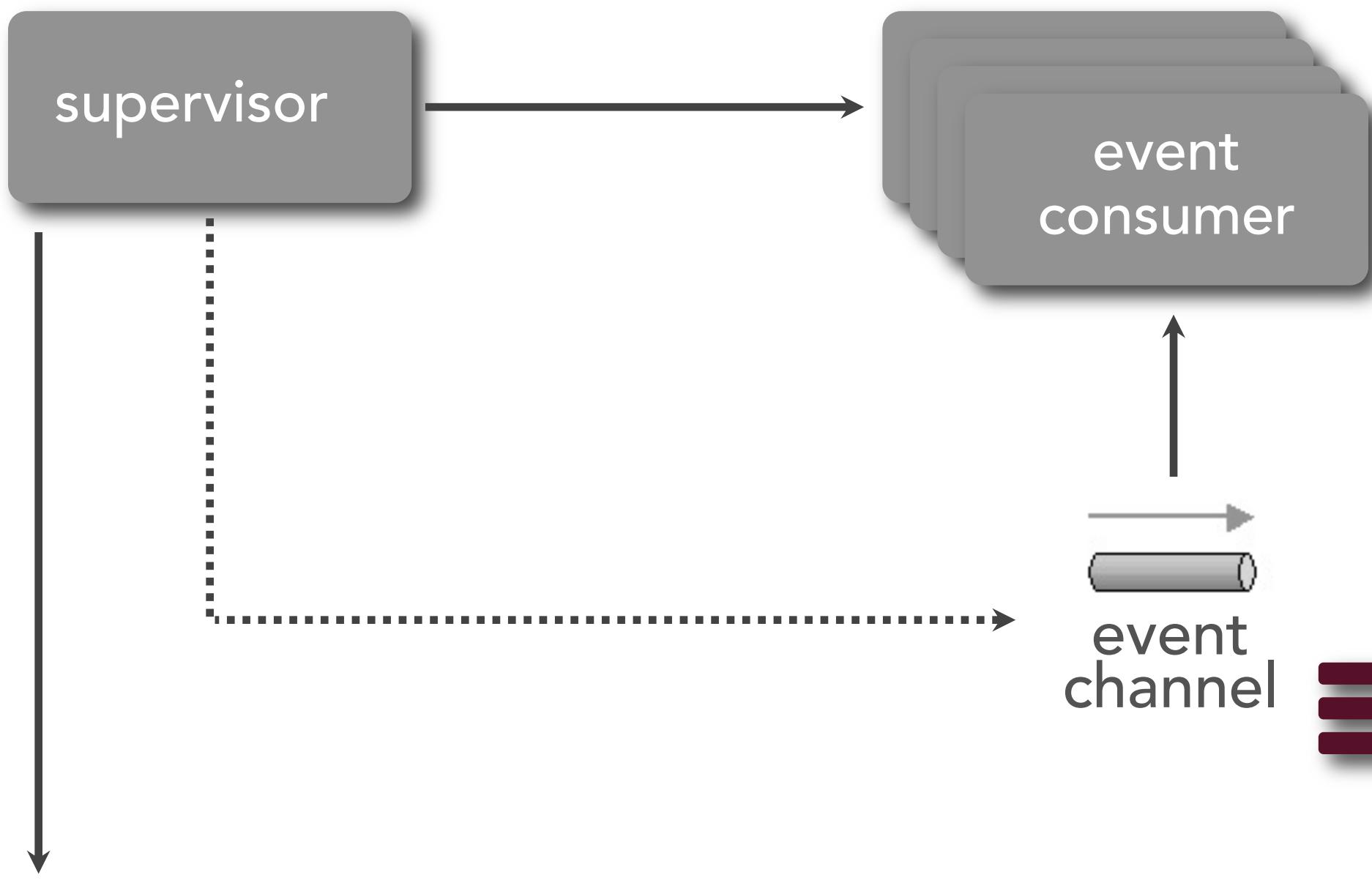
continually monitor queue depth (e.g., 1000ms)  
determine consumers needed ( $\text{depth}/n$ )  
apply max threshold (e.g., 500 consumers)

# supervisor consumer pattern



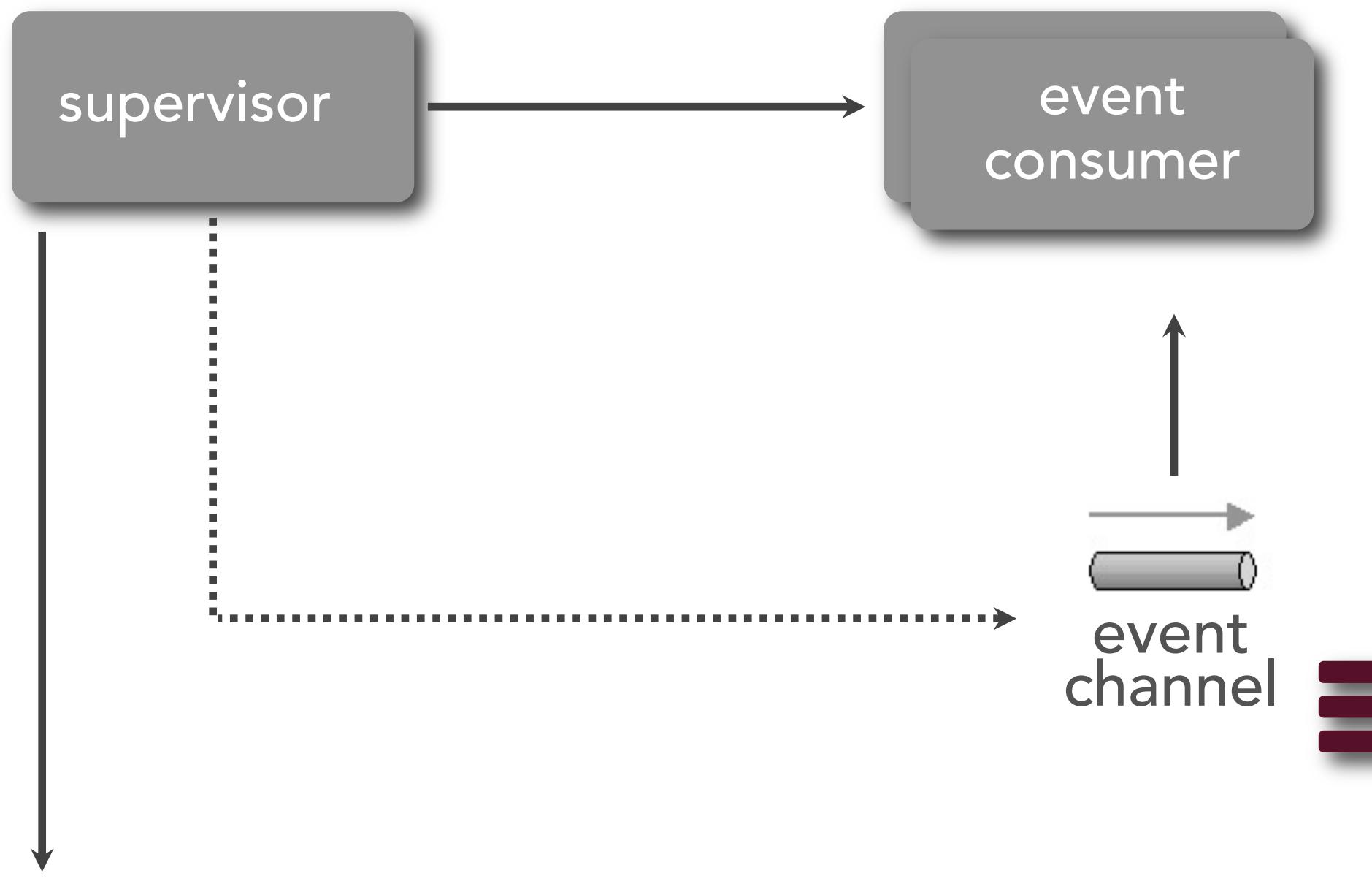
continually monitor queue depth (e.g., 1000ms)  
determine consumers needed ( $\text{depth}/n$ )  
apply max threshold (e.g., 500 consumers)  
add or remove consumers as needed

# supervisor consumer pattern



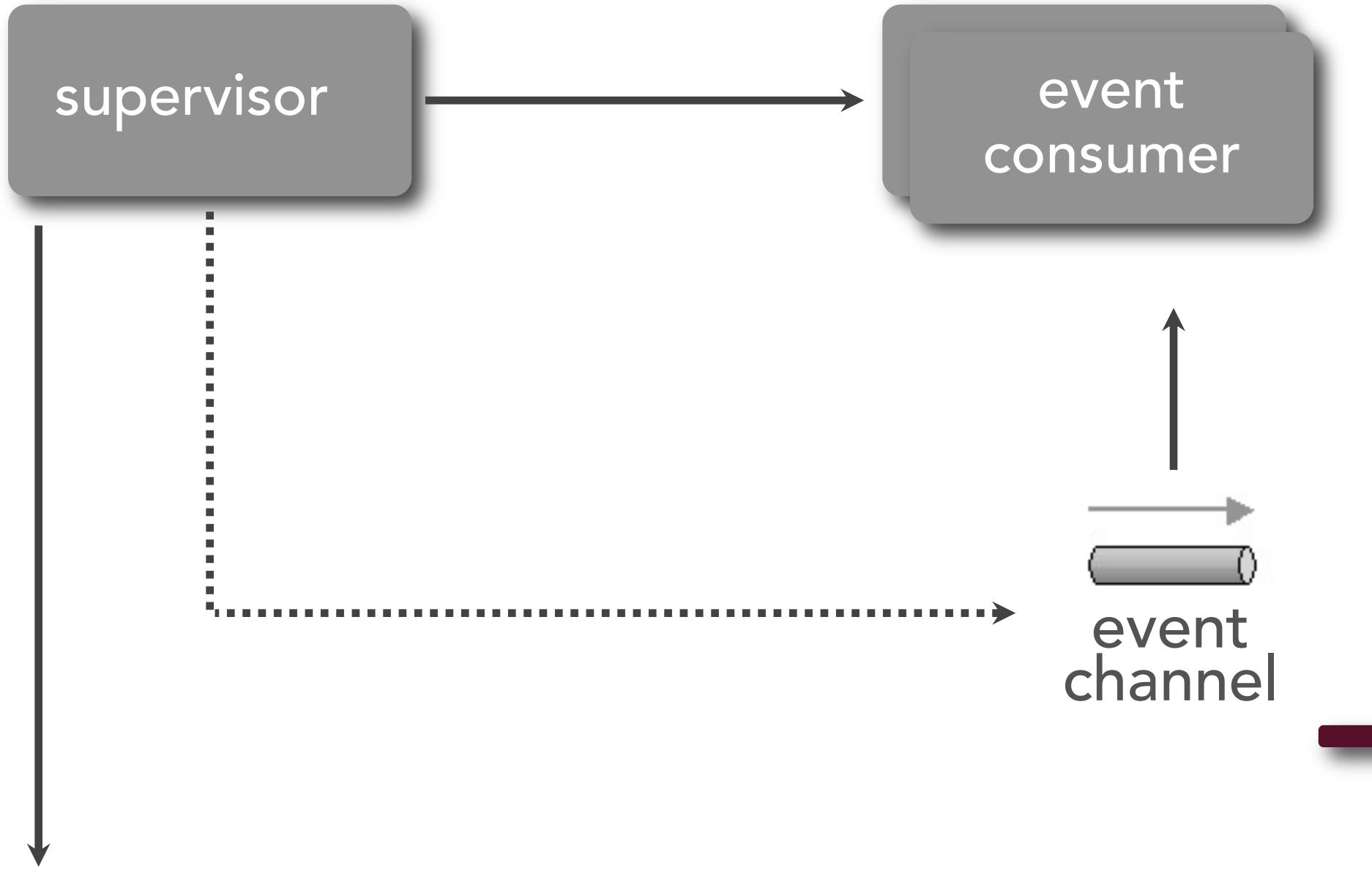
continually monitor queue depth (e.g., 1000ms)  
determine consumers needed ( $\text{depth}/n$ )  
apply max threshold (e.g., 500 consumers)  
add or remove consumers as needed

# supervisor consumer pattern



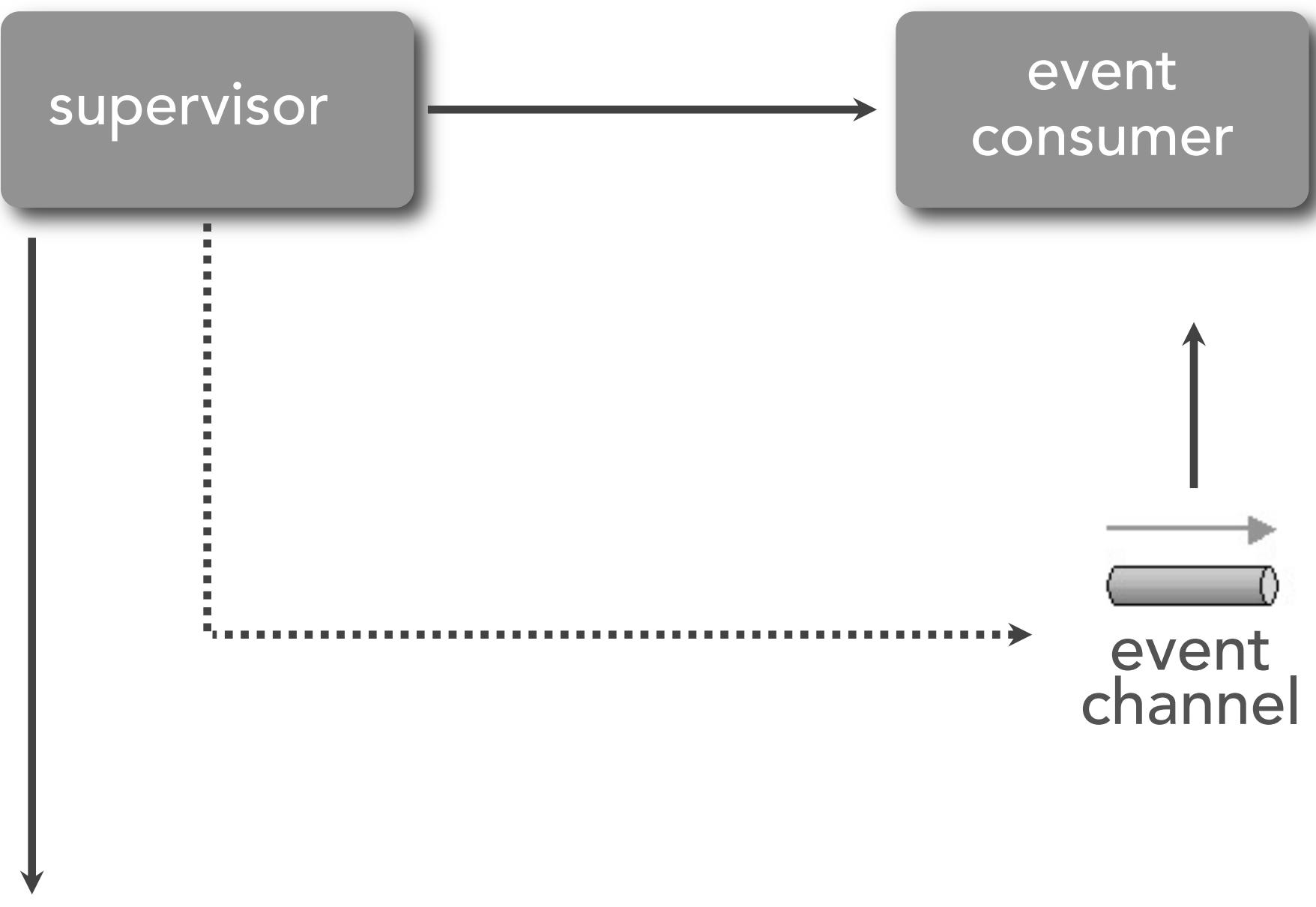
continually monitor queue depth (e.g., 1000ms)  
determine consumers needed ( $\text{depth}/n$ )  
apply max threshold (e.g., 500 consumers)  
add or remove consumers as needed

# supervisor consumer pattern



continually monitor queue depth (e.g., 1000ms)  
determine consumers needed ( $\text{depth}/n$ )  
apply max threshold (e.g., 500 consumers)  
add or remove consumers as needed

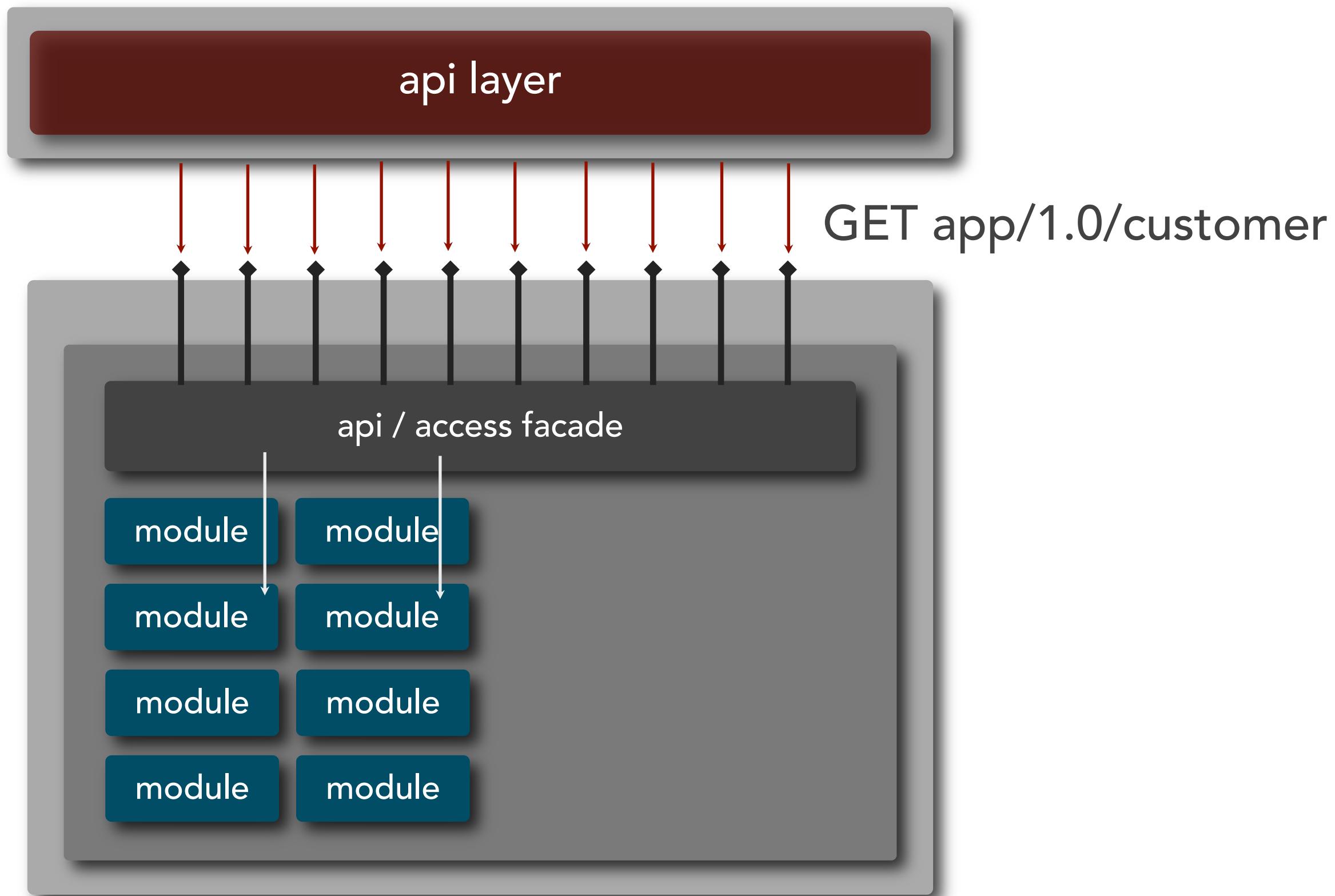
# supervisor consumer pattern



continually monitor queue depth (e.g., 1000ms)  
determine consumers needed ( $\text{depth}/n$ )  
apply max threshold (e.g., 500 consumers)  
add or remove consumers as needed

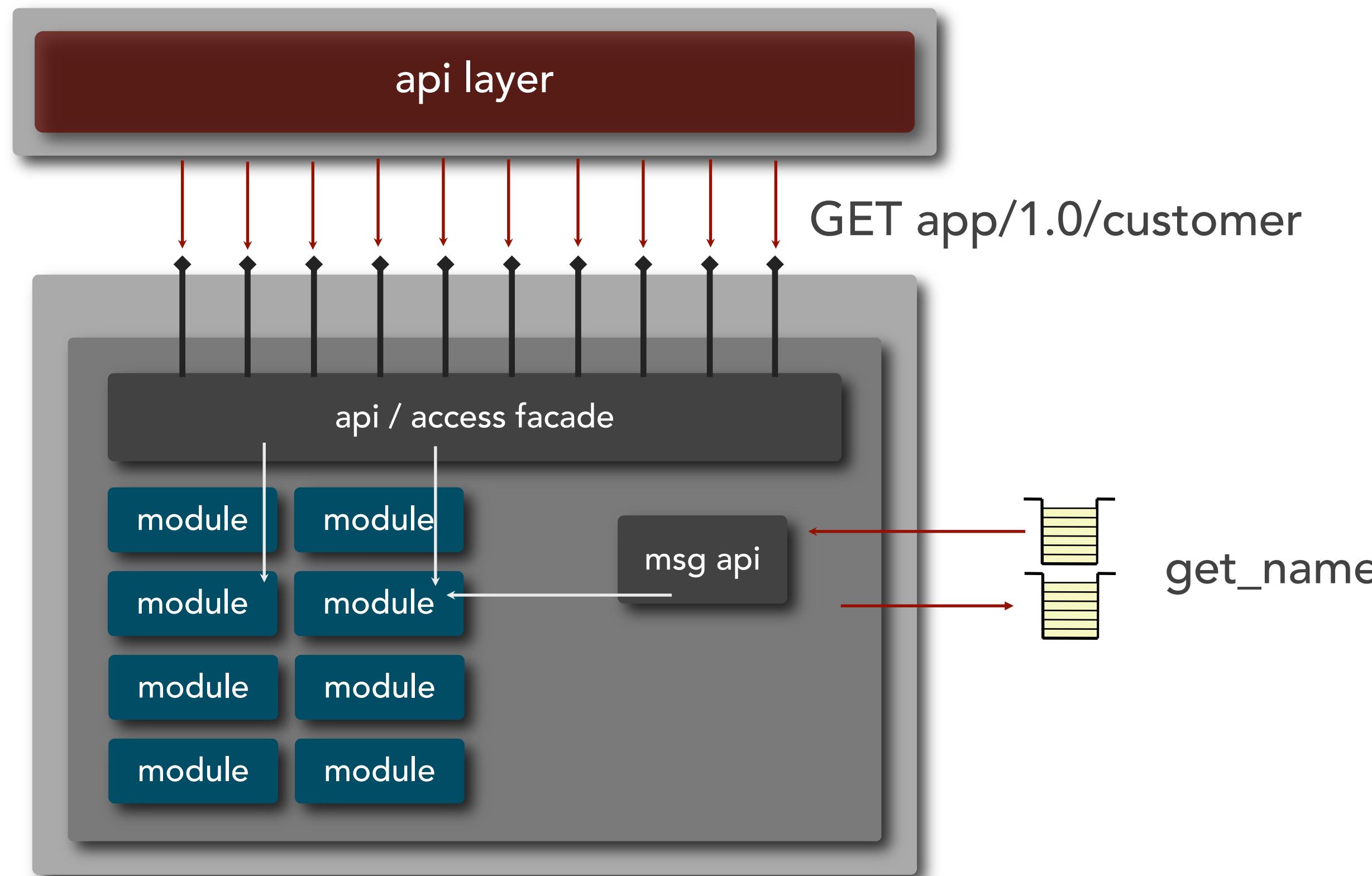
# supervisor consumer pattern

## inter-service communication



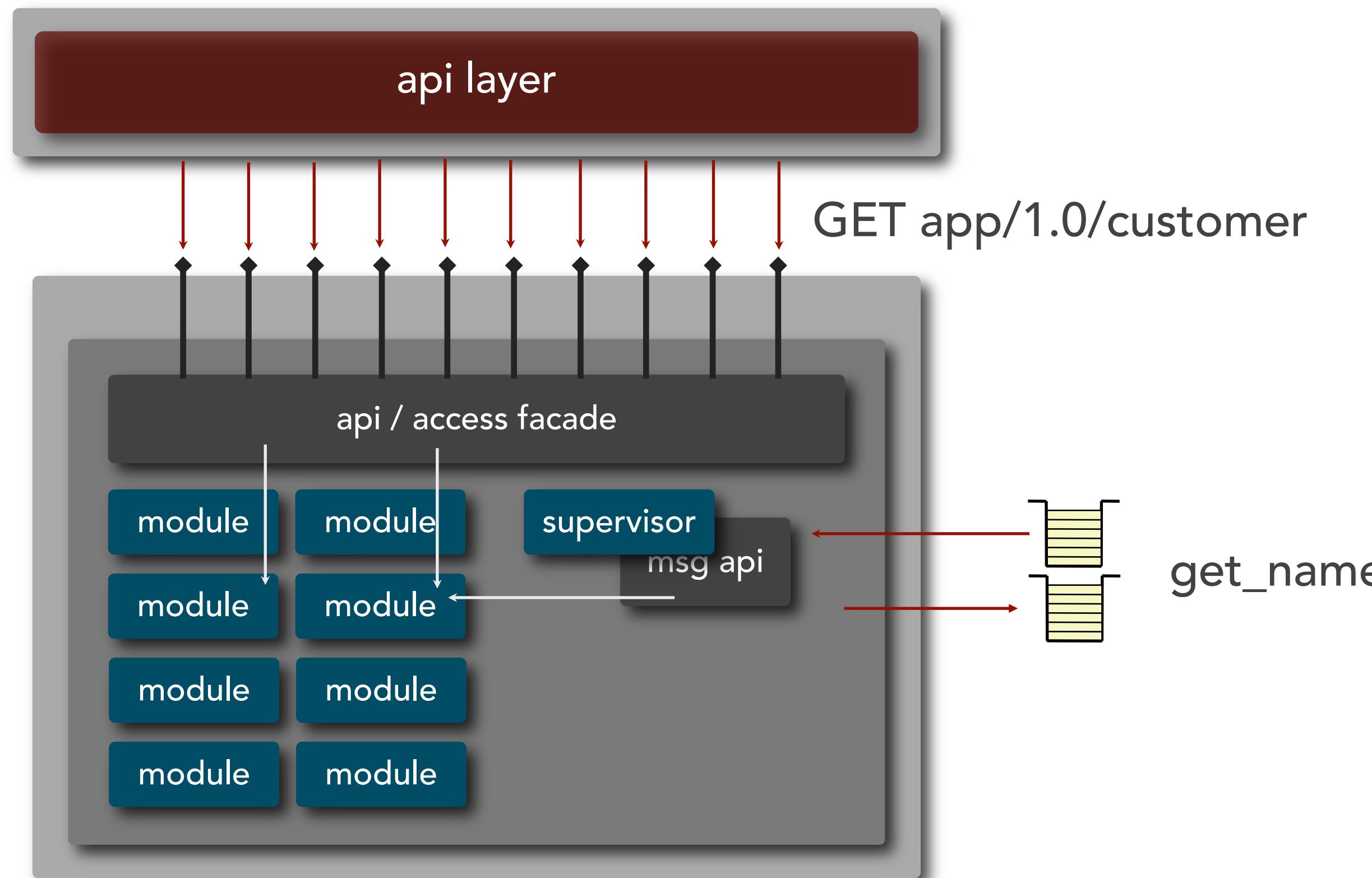
# supervisor consumer pattern

## inter-service communication



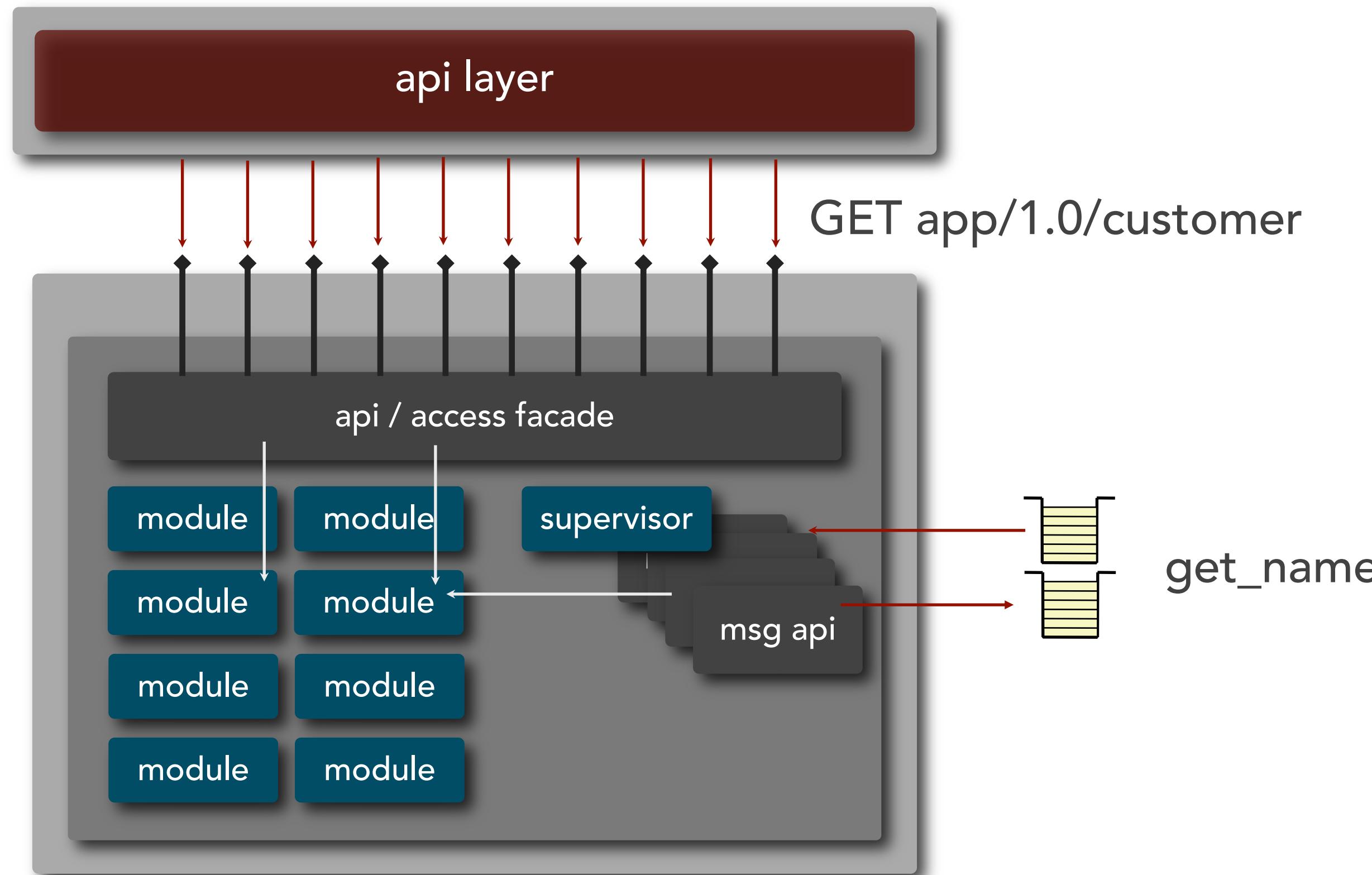
# supervisor consumer pattern

## inter-service communication



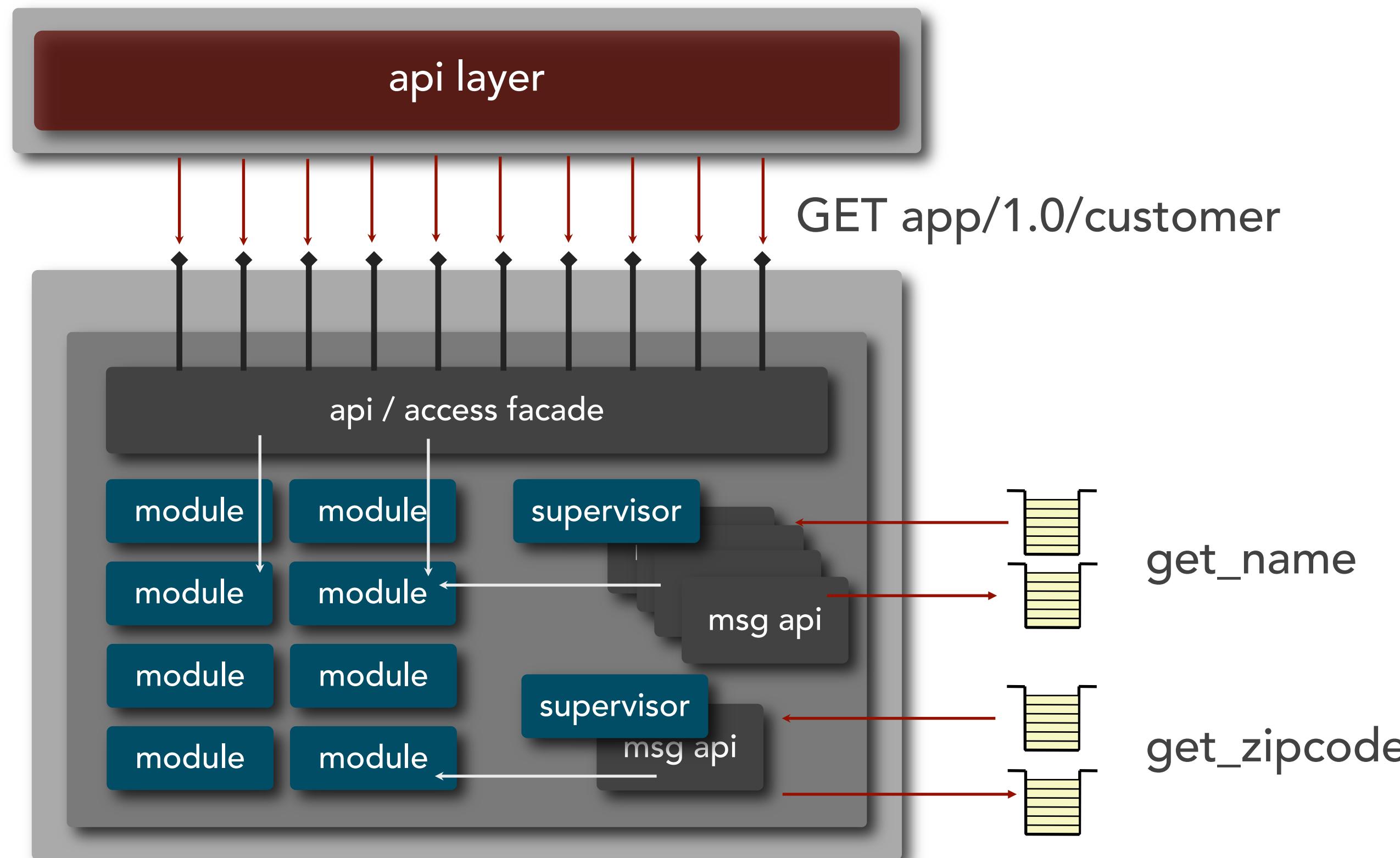
# supervisor consumer pattern

## inter-service communication



# supervisor consumer pattern

## inter-service communication

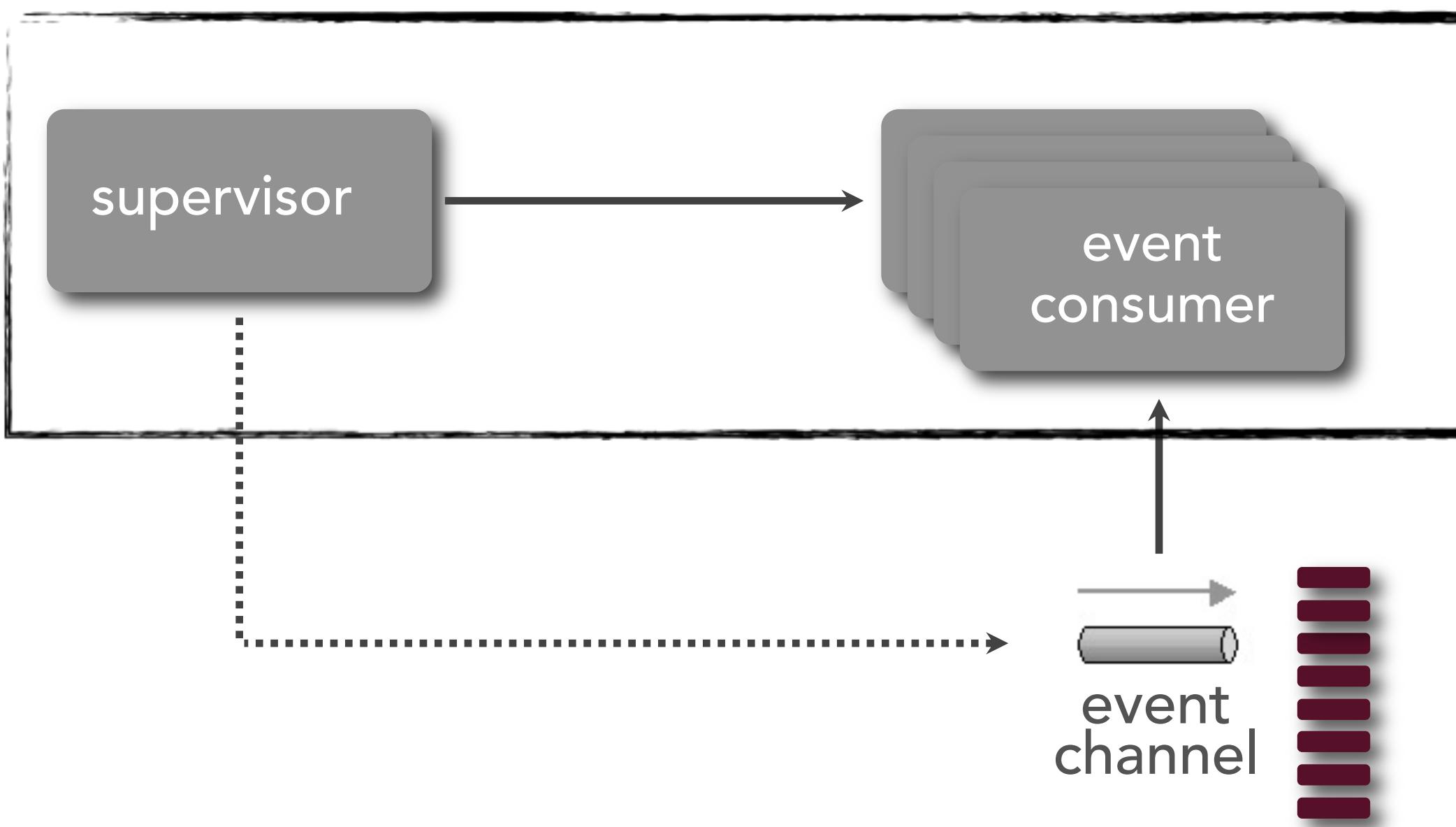


# supervisor consumer pattern

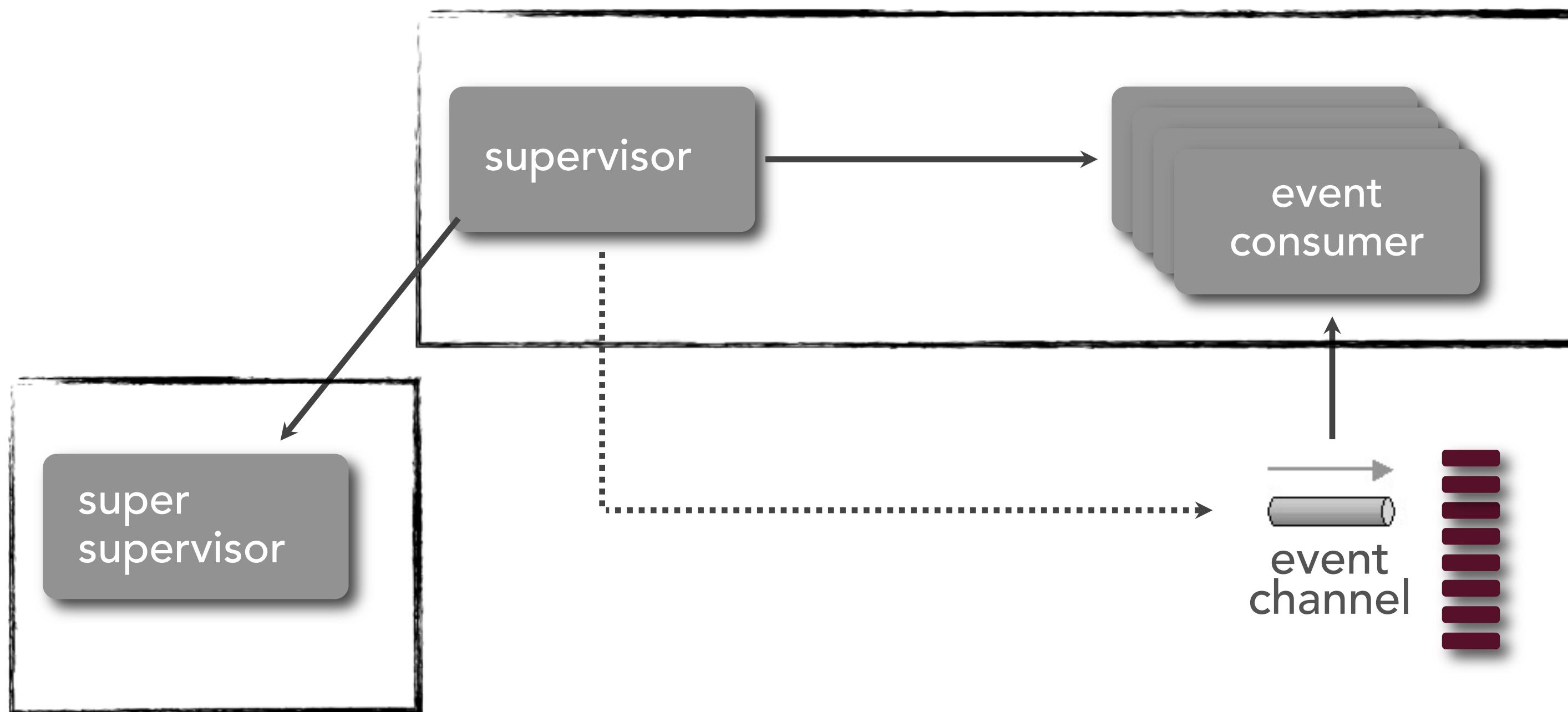


let's see this in action...

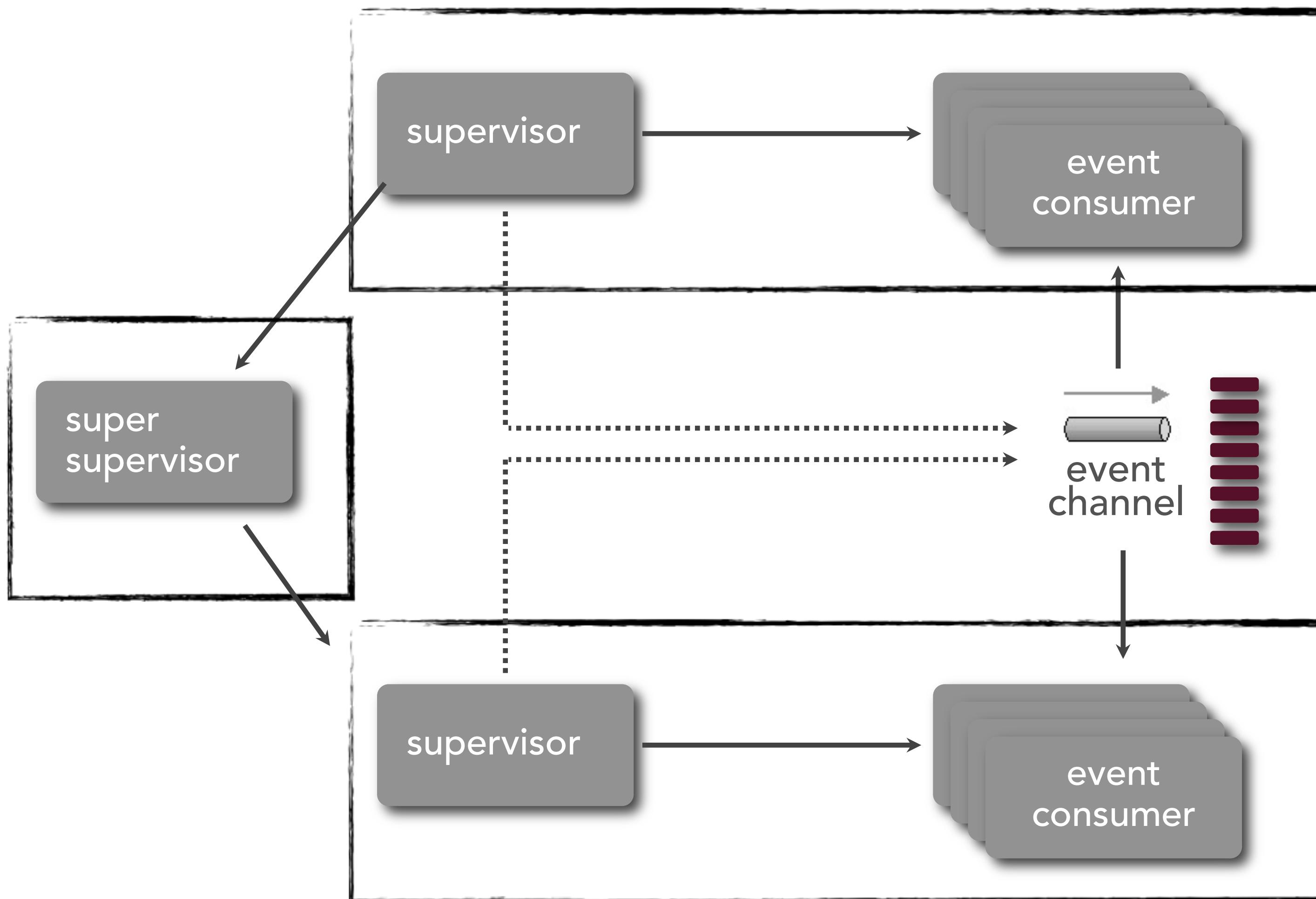
# supervisor consumer pattern



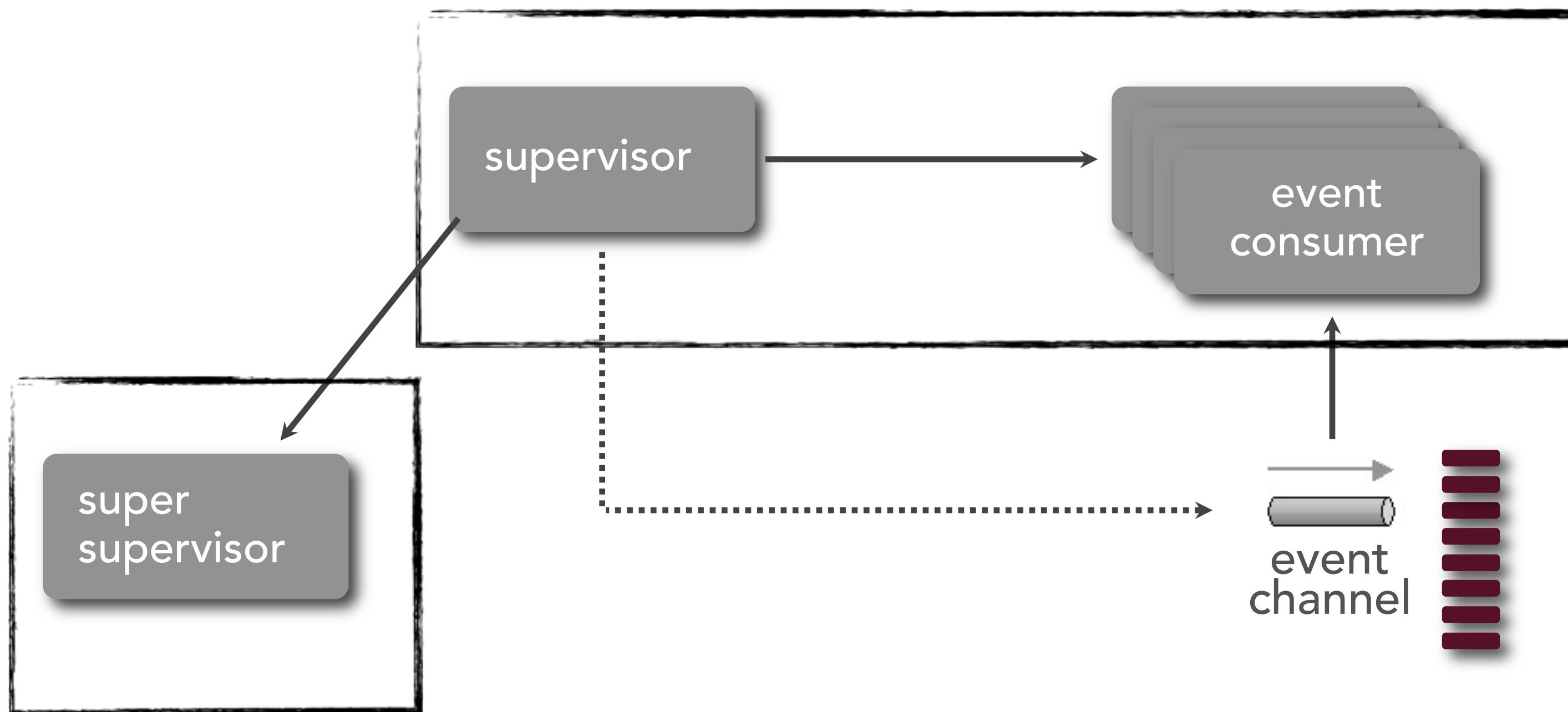
# supervisor consumer pattern



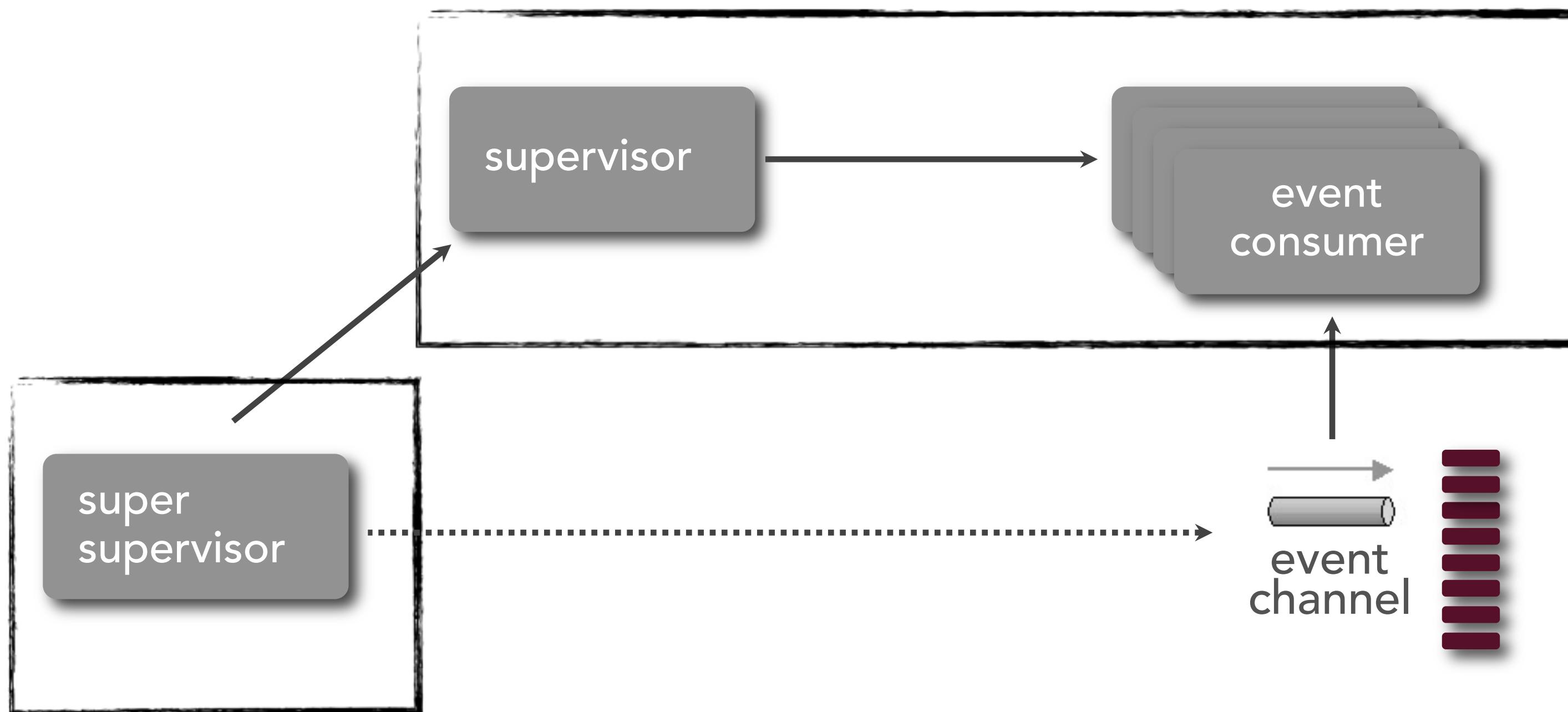
# supervisor consumer pattern



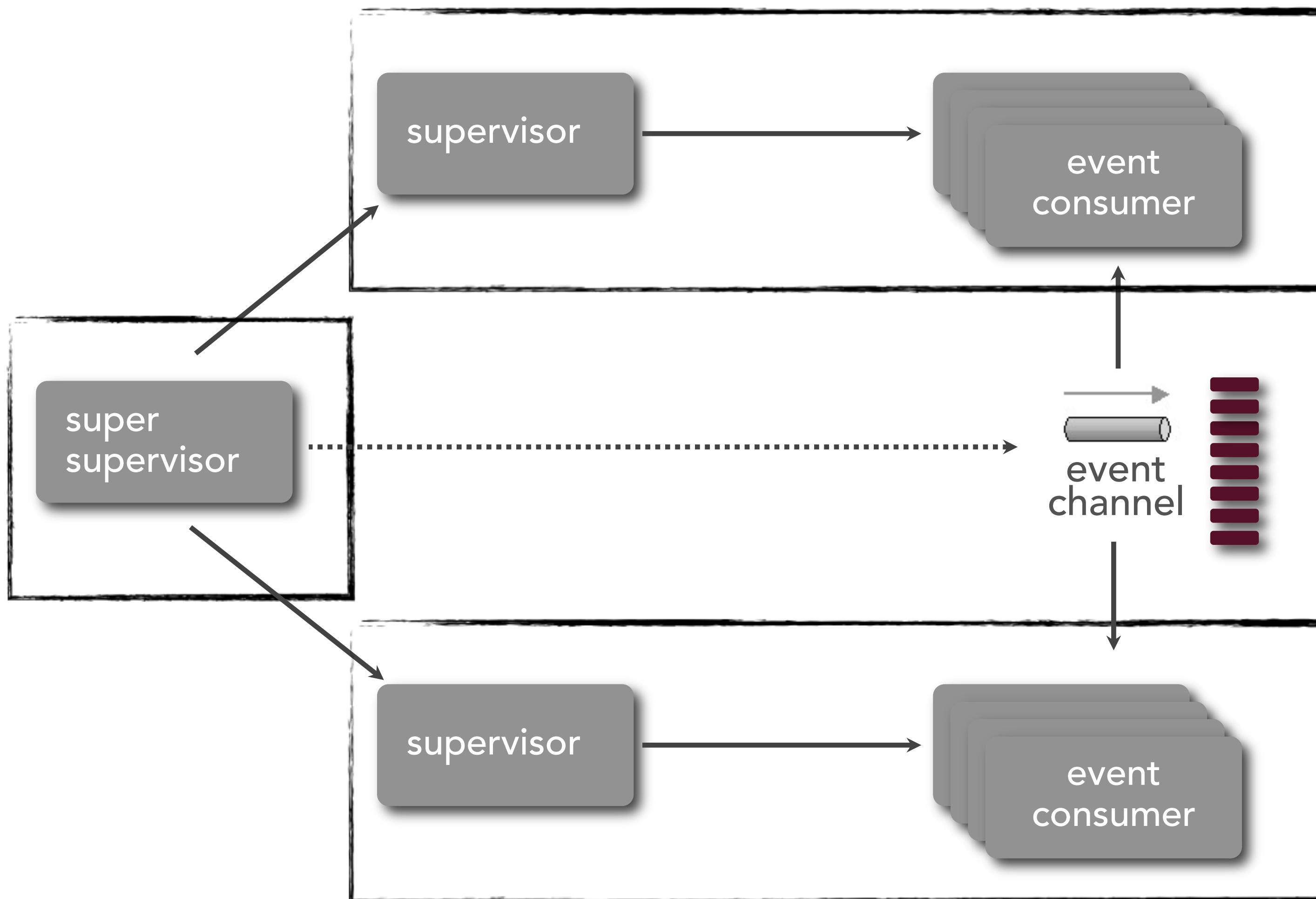
# supervisor consumer pattern



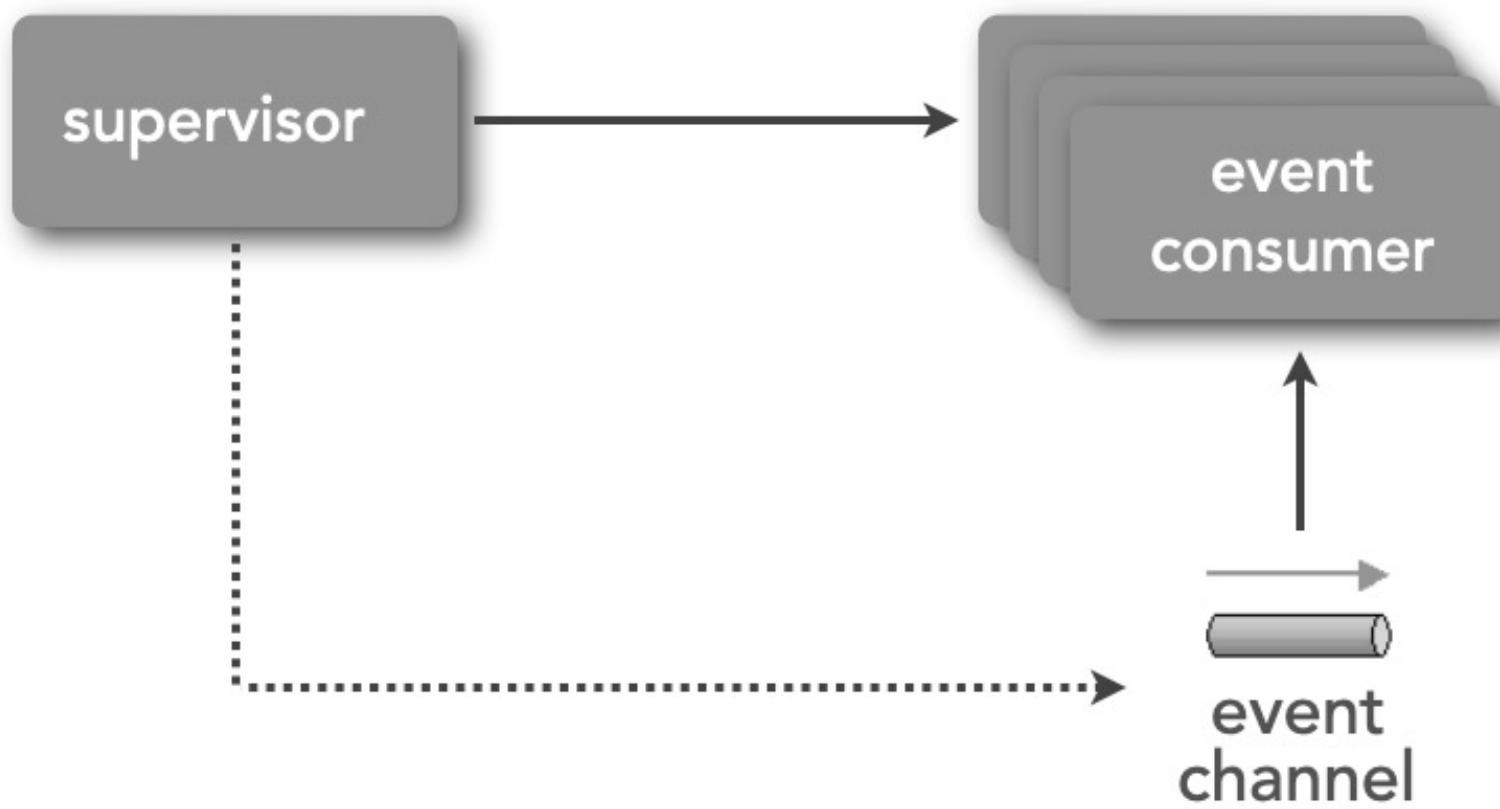
# supervisor consumer pattern



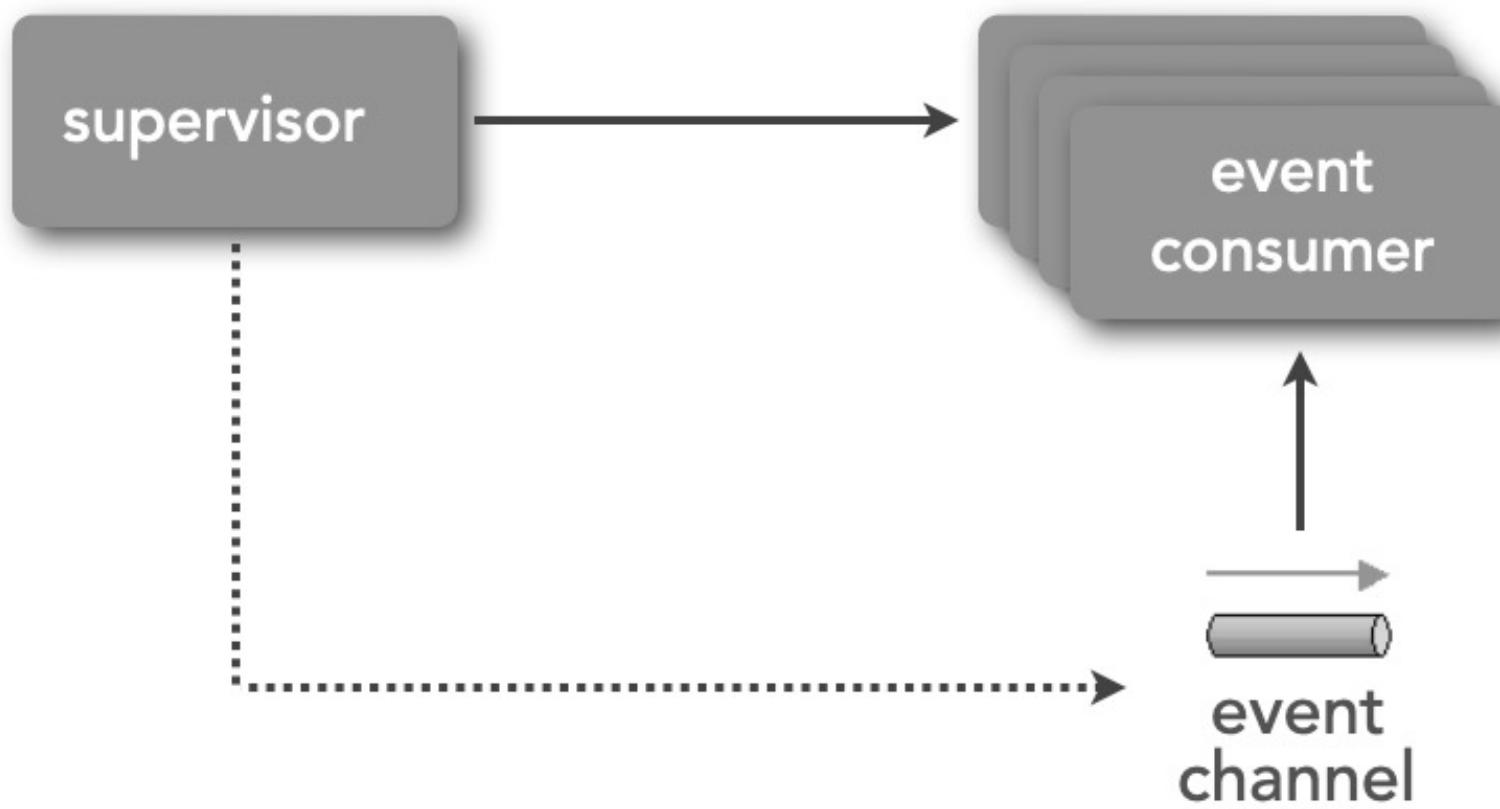
# supervisor consumer pattern



# supervisor consumer pattern



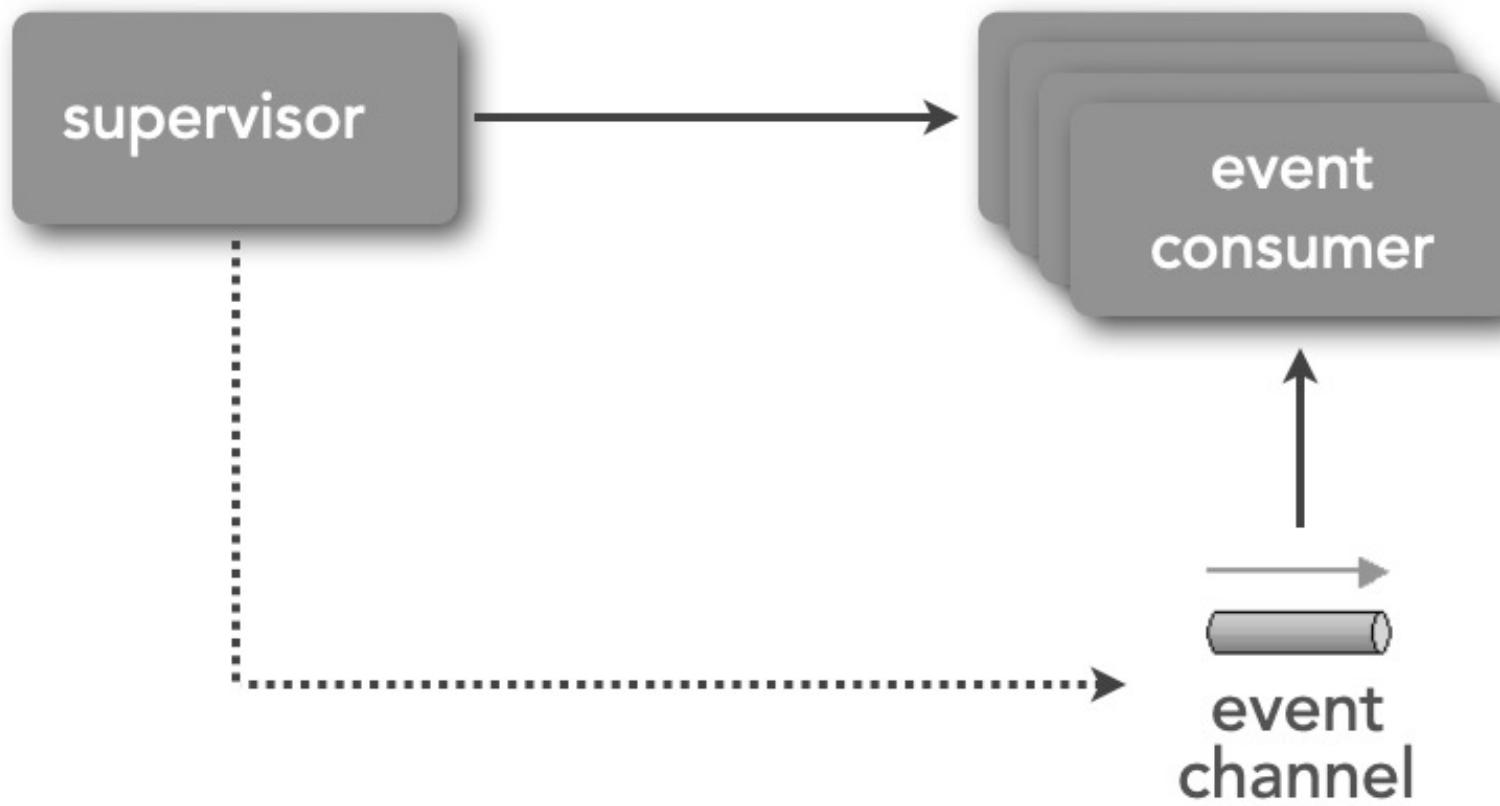
# supervisor consumer pattern



programmatic elasticity  
consistent responsiveness  
optimized thread allocation  
optimized memory usage



# supervisor consumer pattern



programmatic elasticity  
consistent responsiveness  
optimized thread allocation  
optimized memory usage

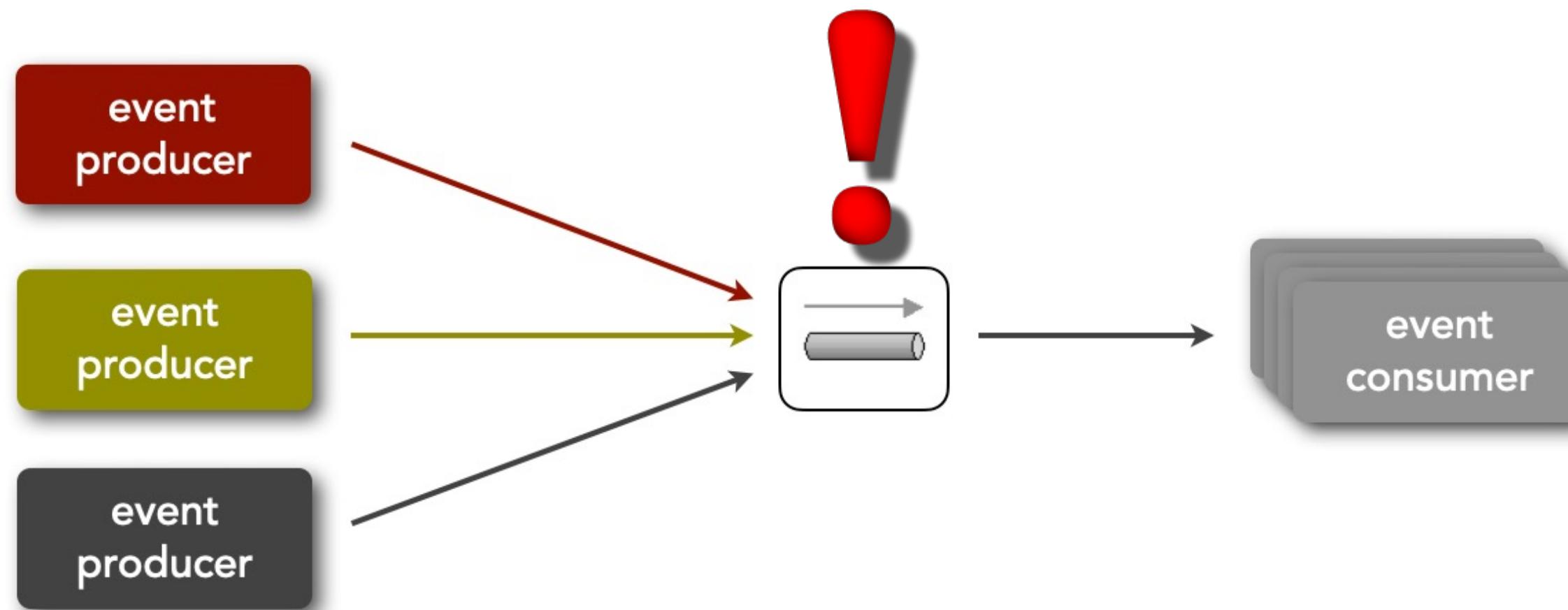


increased complexity  
possible thread saturation  
possible memory errors  
impact on other requests

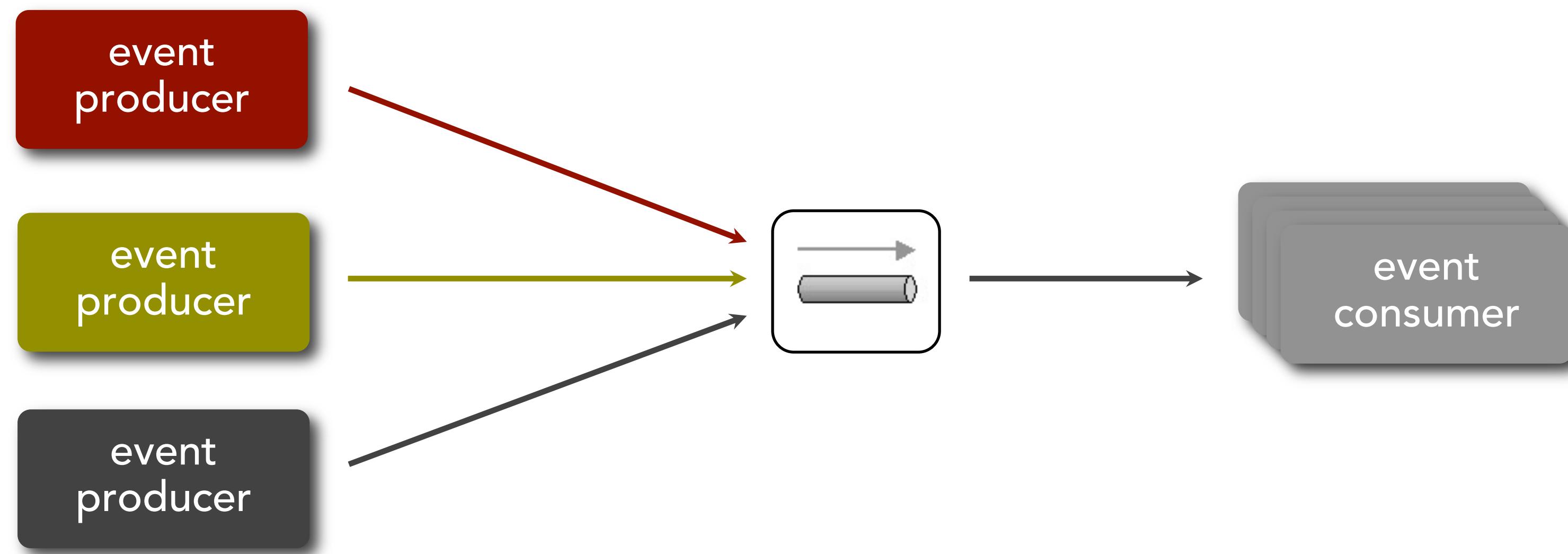
# Multi-Broker Pattern

# multi-broker pattern

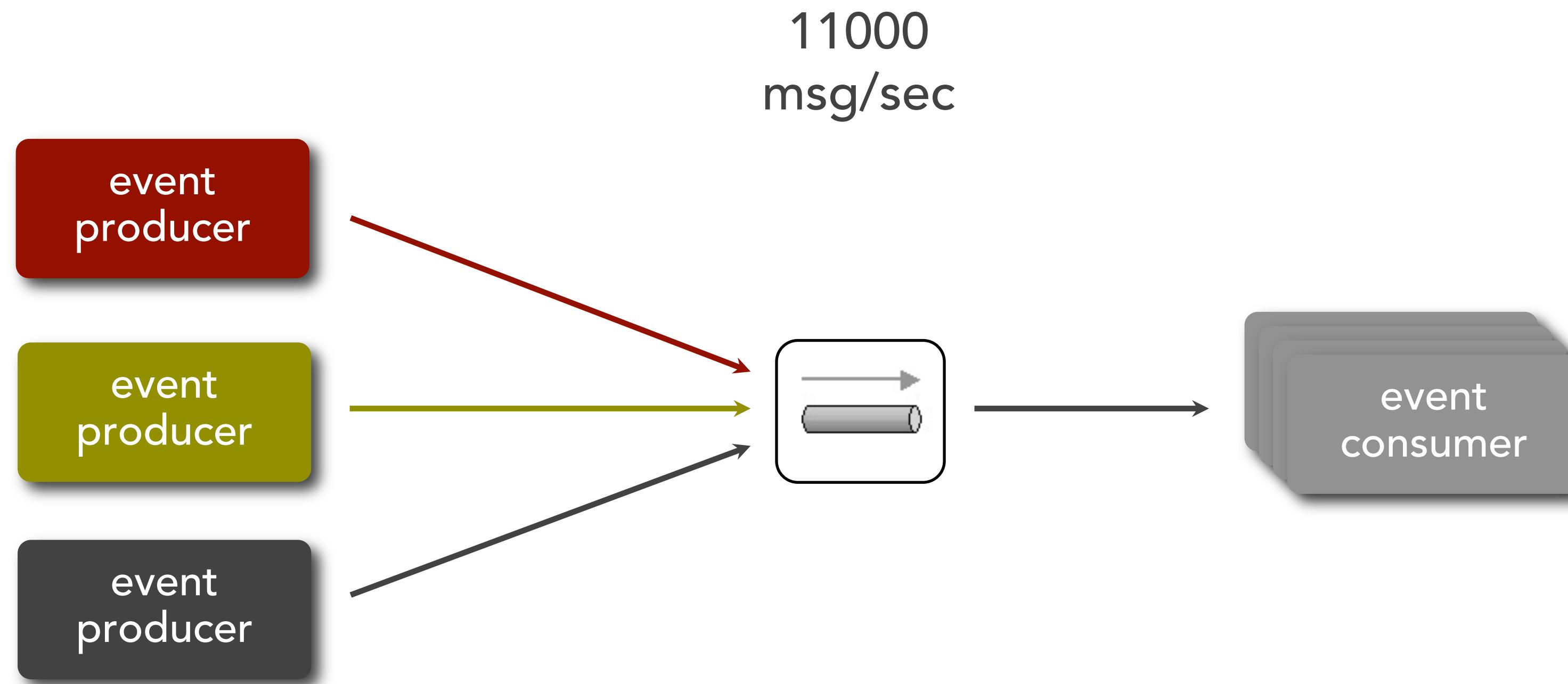
*“how can I increase the throughput and capacity of events through the system?”*



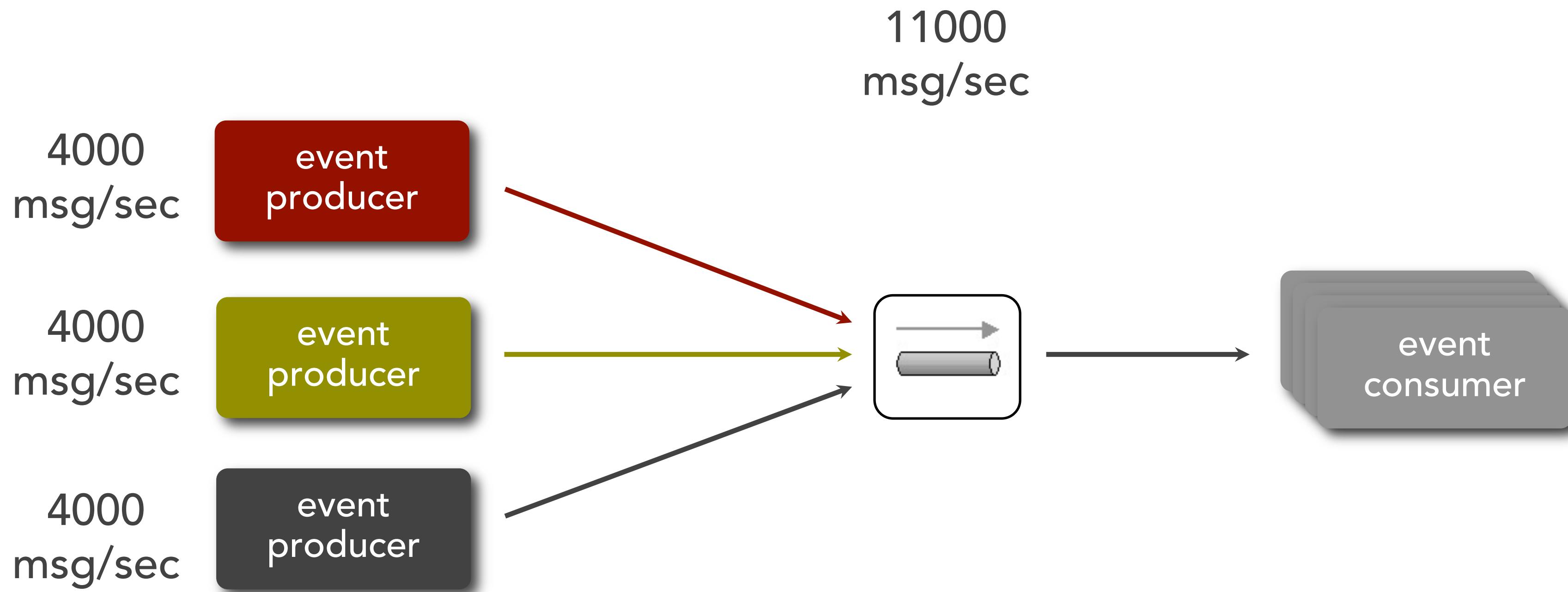
# multi-broker pattern



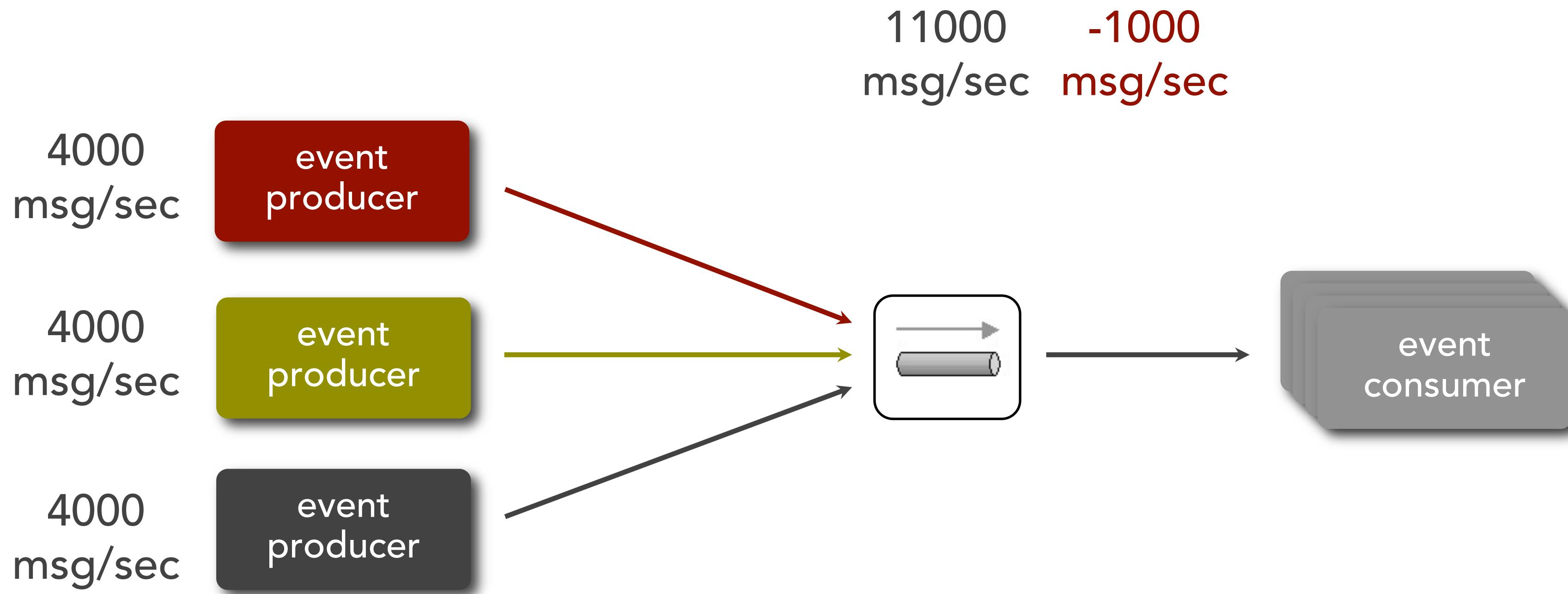
# multi-broker pattern



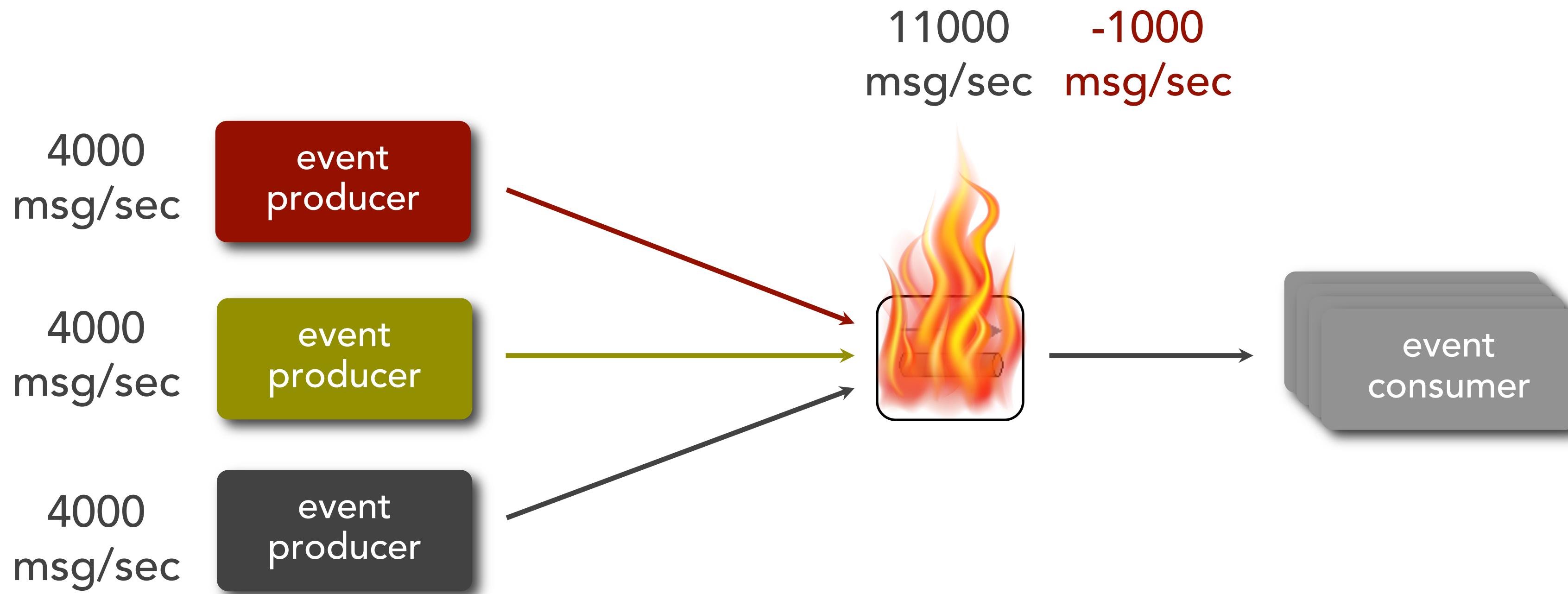
# multi-broker pattern



# multi-broker pattern



# multi-broker pattern



# multi-broker pattern

4000  
msg/sec

event  
producer

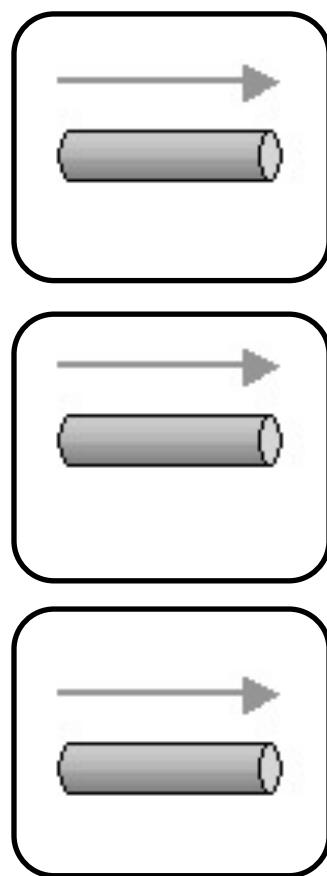
4000  
msg/sec

event  
producer

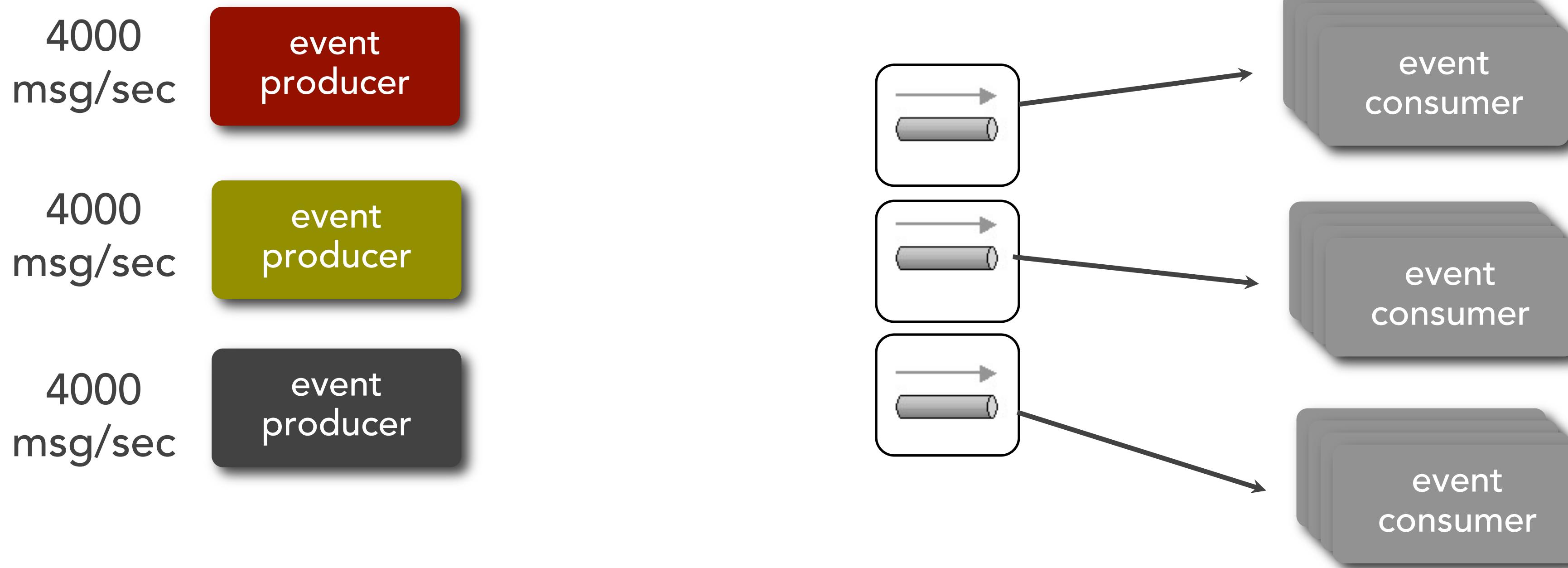
4000  
msg/sec

event  
producer

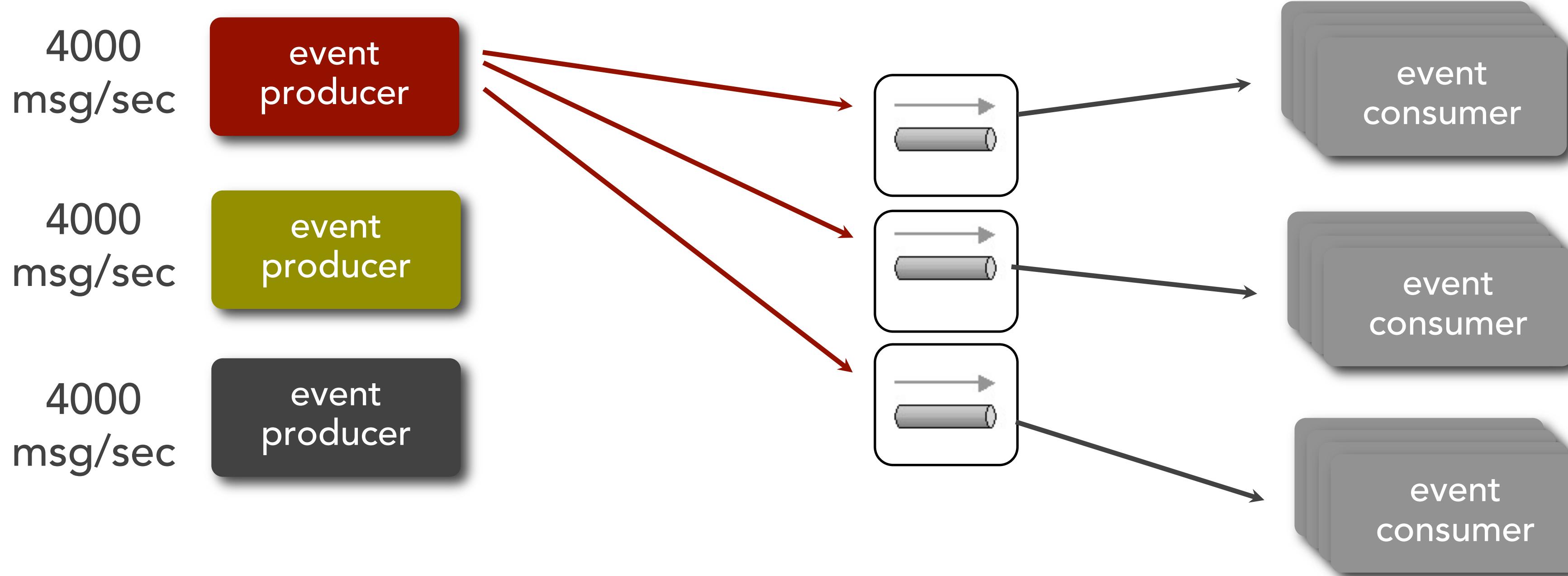
# multi-broker pattern



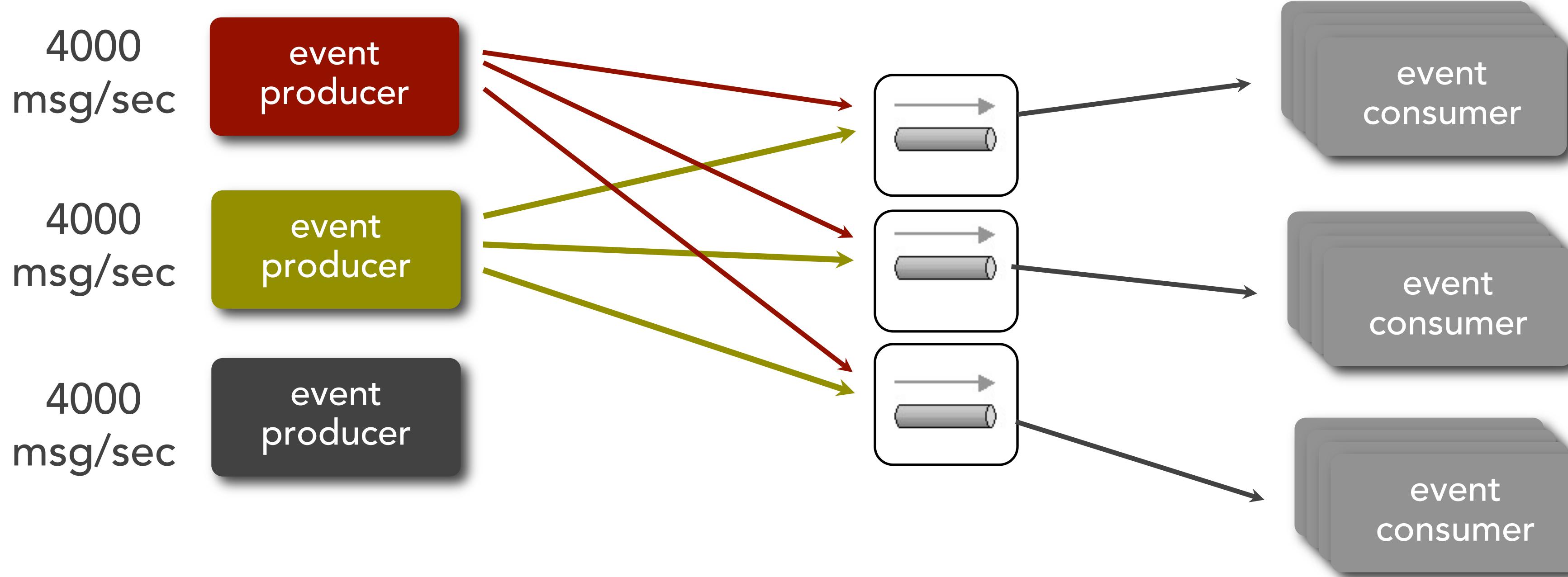
# multi-broker pattern



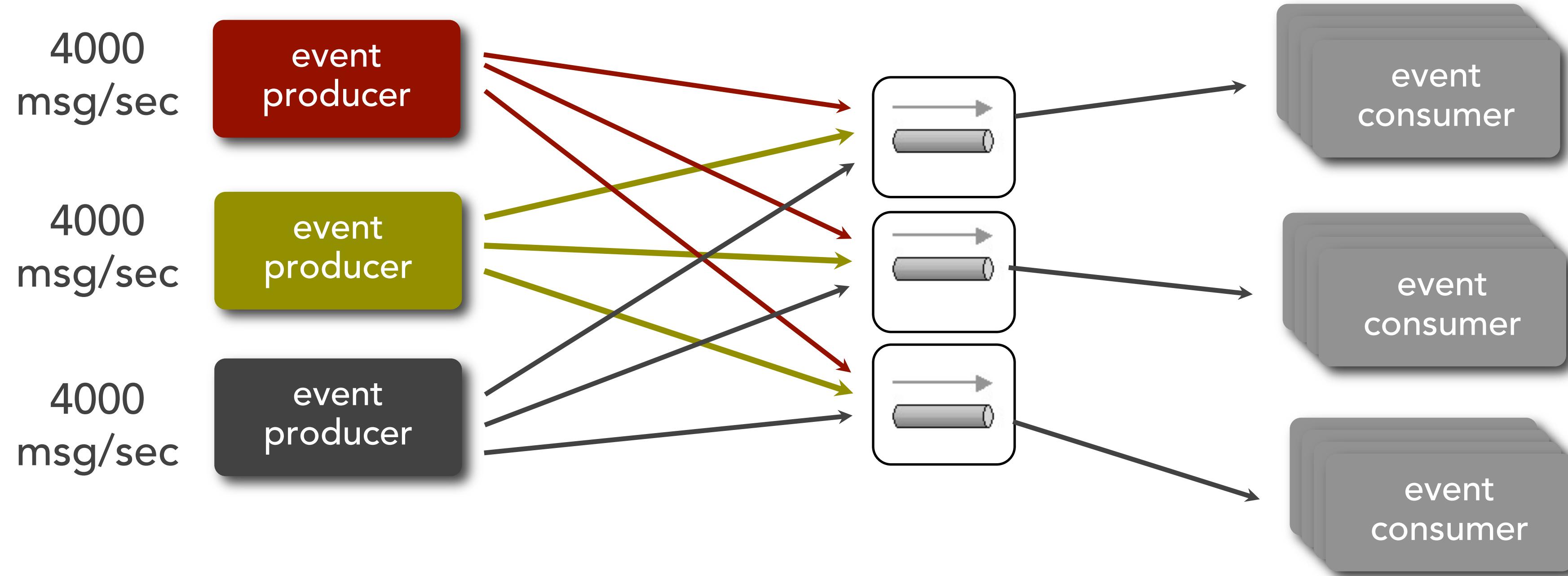
# multi-broker pattern



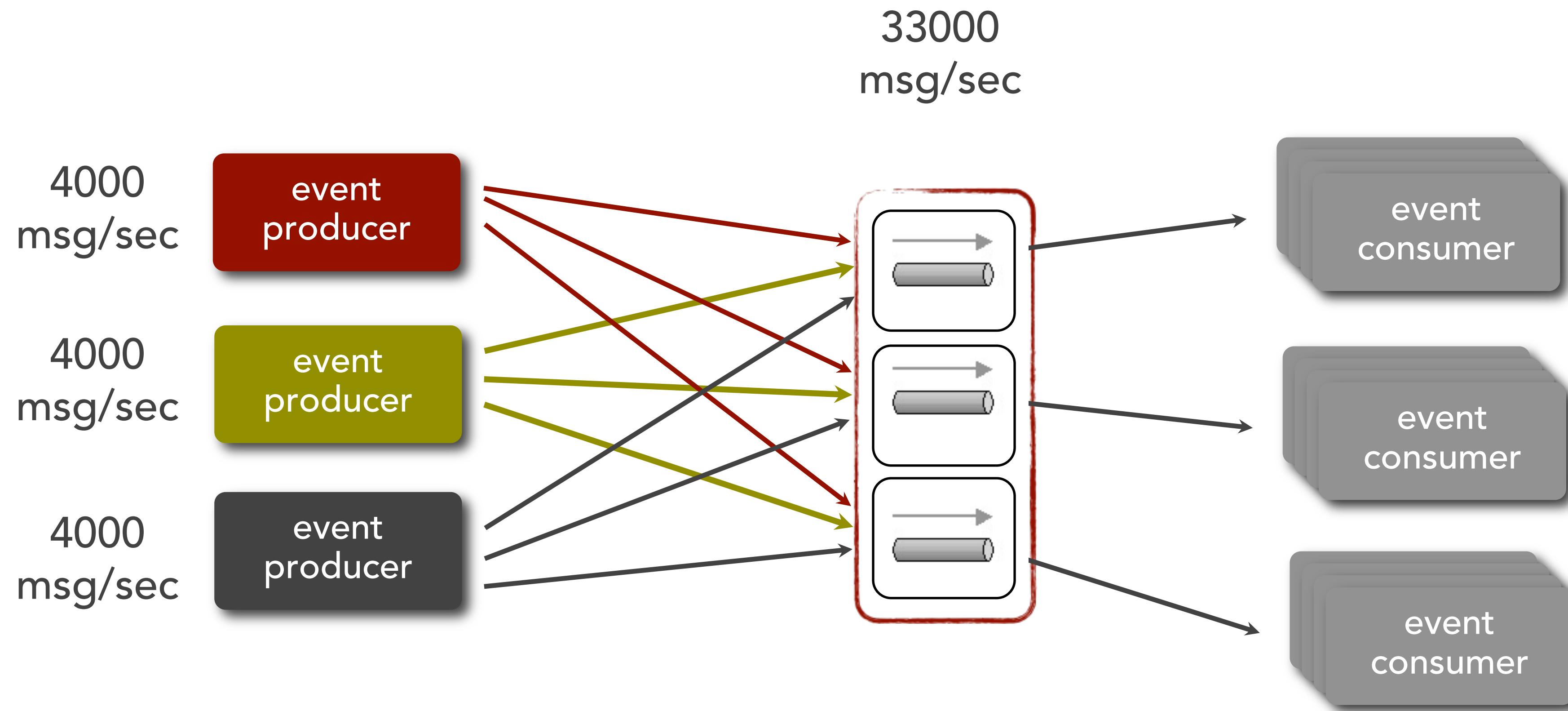
# multi-broker pattern



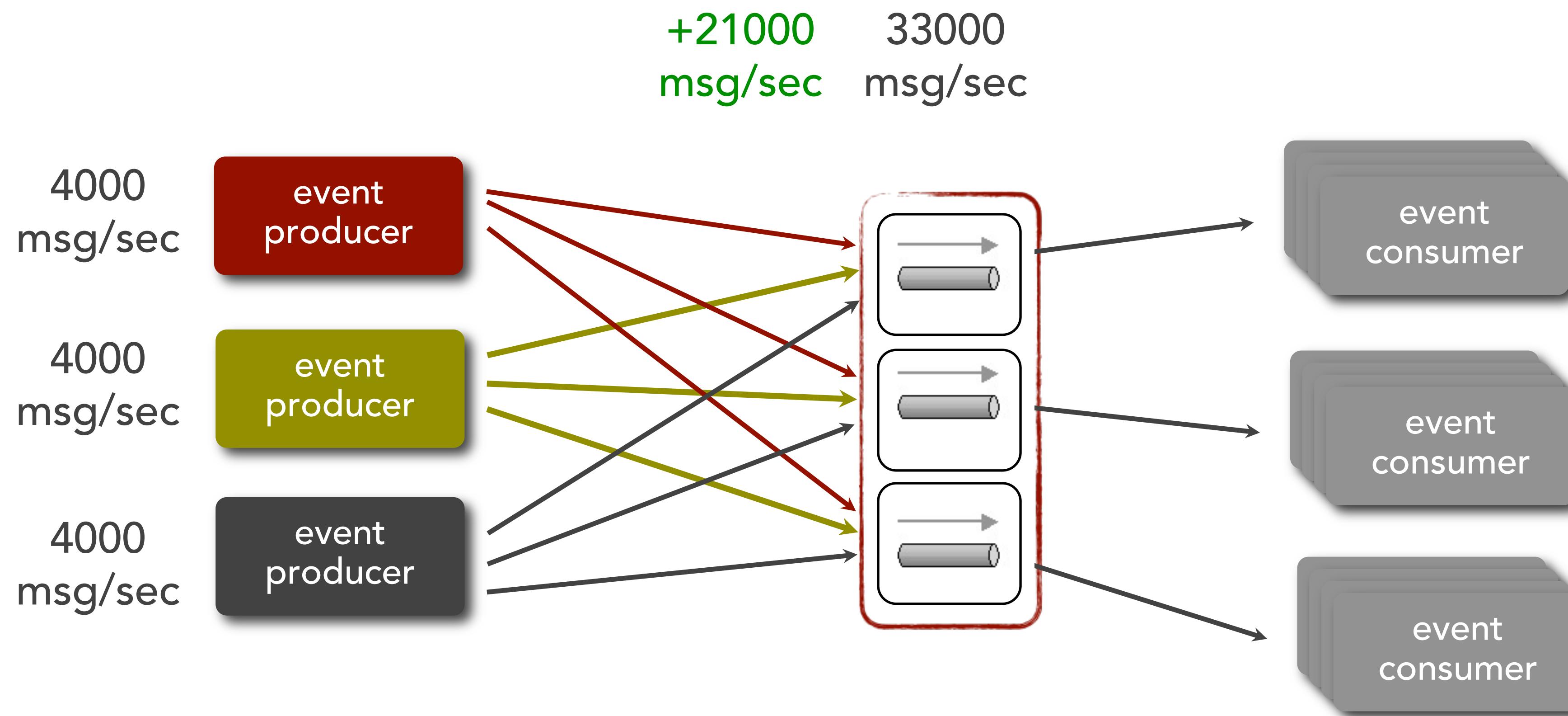
# multi-broker pattern



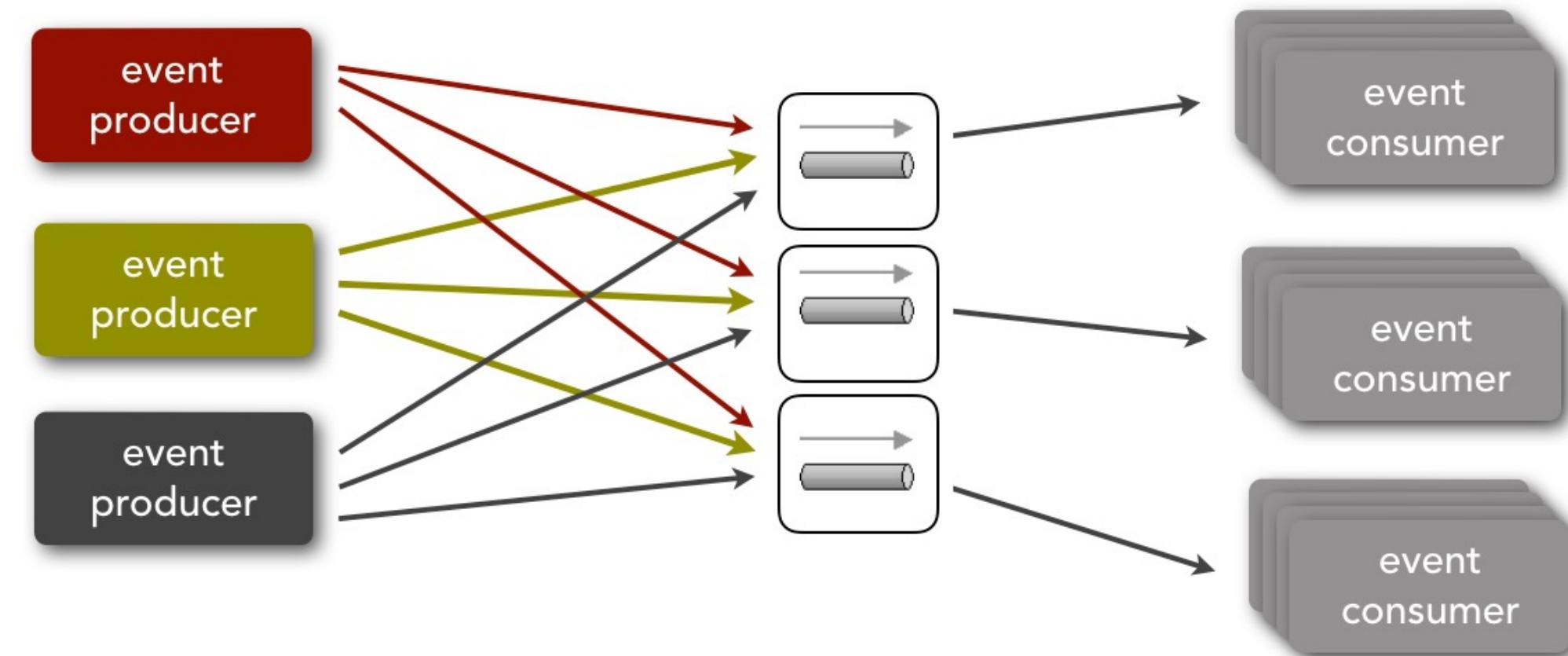
# multi-broker pattern



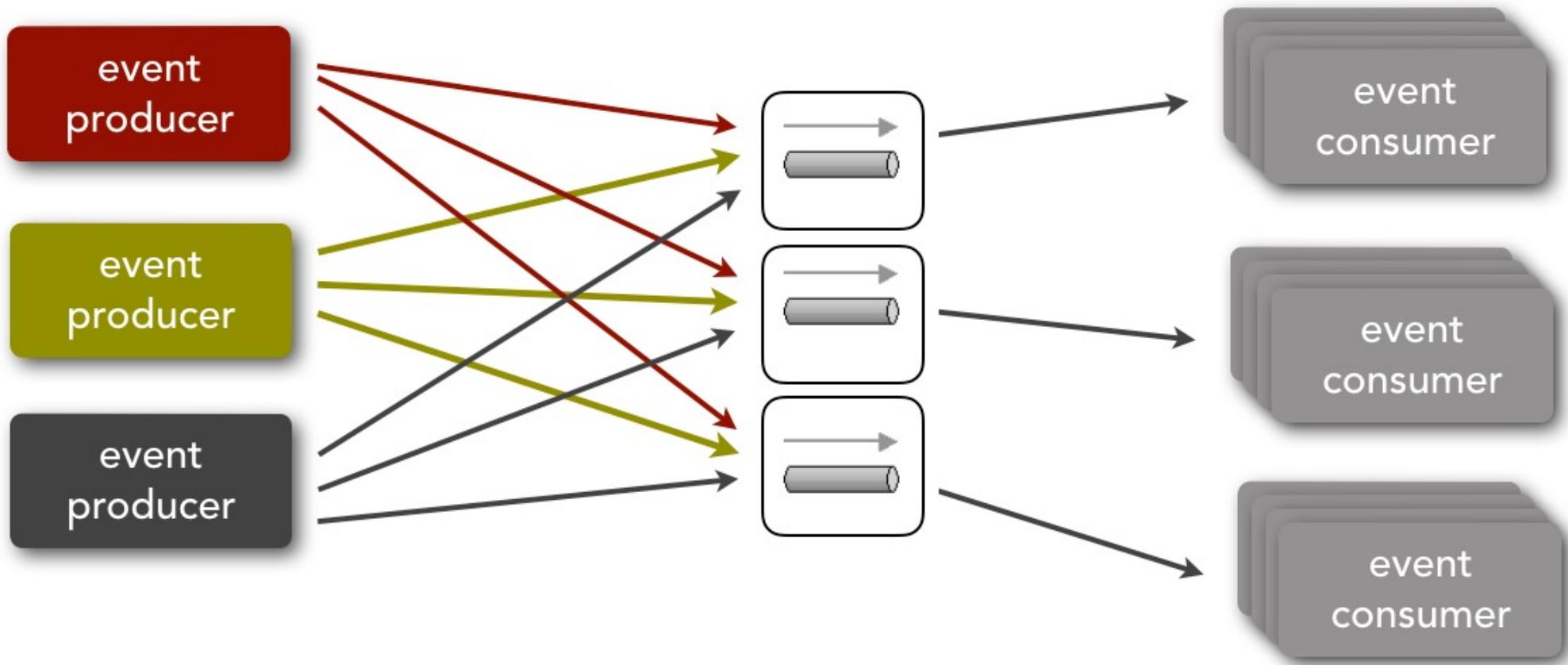
# multi-broker pattern



# multi-broker pattern



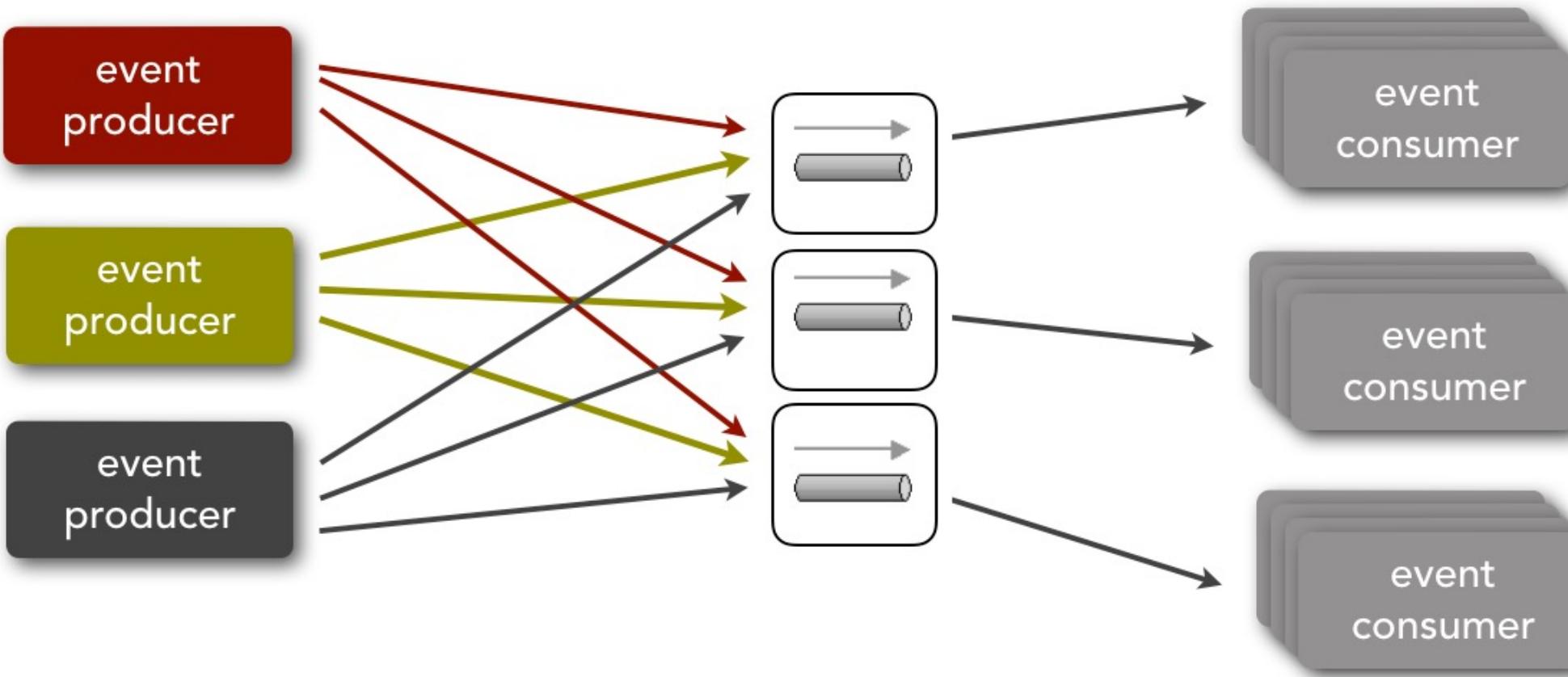
# multi-broker pattern



throughput  
performance  
scalability  
capacity



# multi-broker pattern



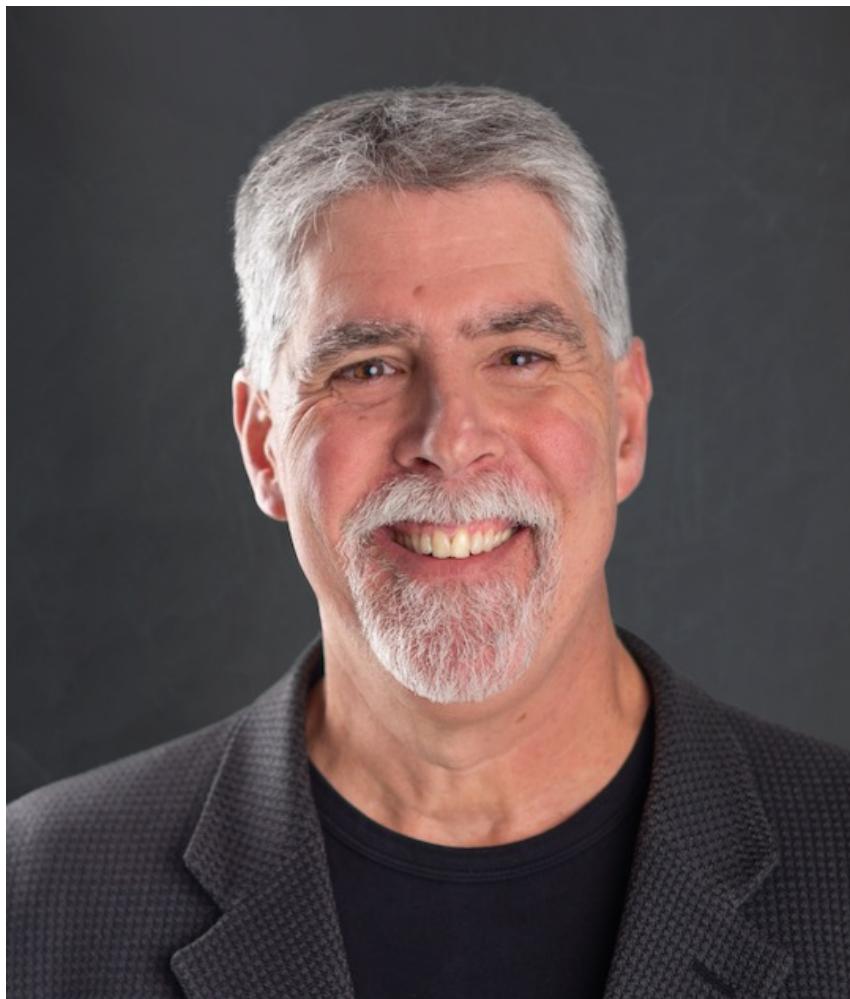
throughput  
performance  
scalability  
capacity



complexity  
cost  
fifo message order



# Mastering Patterns in Event-Driven Architecture



**Mark Richards**

**Independent Consultant**

**Hands-on Software Architect / Published Author**

**Founder, DeveloperToArchitect.com**

**<https://www.linkedin.com/in/markrichards3>**

**@markrichardssa**